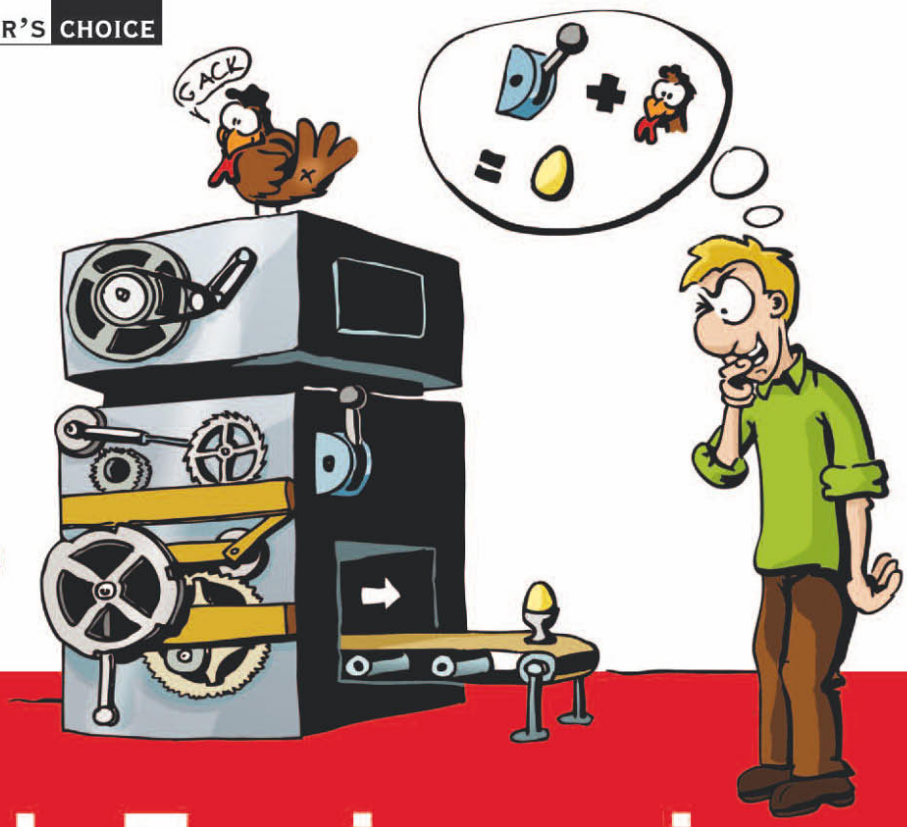
PROGRAMMER'S CHOICE

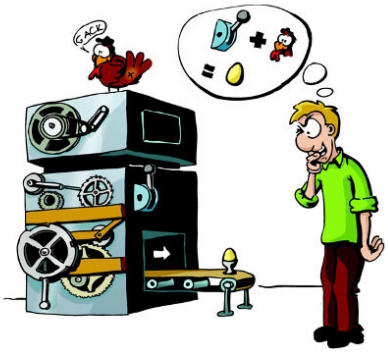


Sven Busse

Flash Engineering

Agile Ansätze zum Bau von RIAs
mit Flash, Flex und ActionScript

ADDISON-WESLEY



3 Lösungen entwerfen

Einer der großen Stärken der Menschen ist es, als Vernunftwesen Zusammenhänge allein im Kopf zu durchdenken und daraus abstrakte Erkenntnisse zu gewinnen. Wir können aus konkreten beobachteten Vorgängen Rückschlüsse auf zugrunde liegende Zusammenhänge schließen, ohne sie zwingend durch konkrete Erfahrung erlangt zu haben. Wenn wir einmal gesehen haben, wie ein Tennisball vom Boden oder einer Wand abprallt, beginnen wir, in Gedanken die Laufbahn des Balles vorauszuahnen, wenn wir ihn in eine bestimmte Richtung werfen würden, ohne es tatsächlich tun zu müssen. Wir sind außerdem in der Lage, von konkreten Dingen zu abstrahieren, und wir erkennen somit, dass sich mit großer Wahrscheinlichkeit jeder ballrunde Gegenstand, der aus einem elastischen Material besteht, ähnlich zu dem Tennisball verhalten wird.

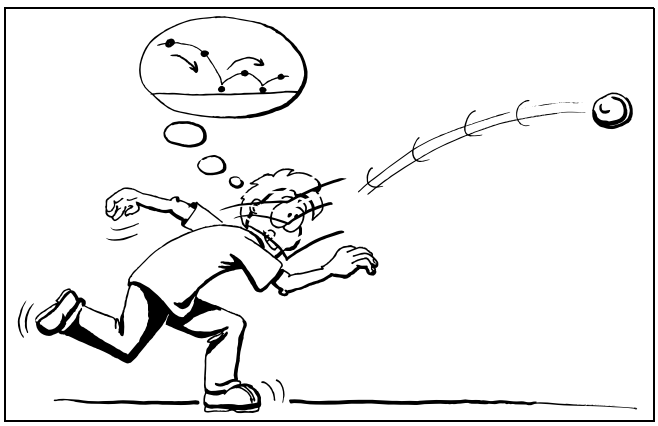


Abbildung 3.1: Wir haben eine ungefähre Vorstellung davon, wie der Ball fliegen wird.

Diese Fähigkeit hilft uns in fast allen Lebenslagen, und sie wird in der Softwareentwicklung gleichfalls enorm gefordert. Ein Softwareentwickler muss Vorgänge und Zusammenhänge aus der realen Welt erkennen und verstehen und sie in einer virtuellen Welt zumeist in abstrakter Weise nachbilden. Neben den vielen Herausforderungen in allen Bereichen der Softwareentwicklung sind das Verstehen und Abstrahieren einer realen Problemstellung eine besonders schwierige Aufgabe. Sie ist es auch deshalb, weil jede Problemstellung in ihren Details meist individuell ist und nicht direkt vergleichbar mit vorangegangenen Projekten.

Der Bau von Software wird hier gerne mit dem Bau von Gebäuden verglichen. Wir können uns so den Bau von Software als das langsame Errichten eines Gebäudes vorstellen, angefangen von den ersten Gesprächen mit dem Bauherren, der Arbeit des Architekten, der Bauingenieure und letztlich der Maurer, Elektriker und der vielen anderen Fachleute, die am Bau eines Gebäudes beschäftigt sind. Viele Parallelen lassen sich zwischen der Softwareentwicklung und dem Gebäudebau erkennen, gerade was Prozesse angeht. Der Vergleich hinkt aber auch. In der Softwareentwicklung wird in der Regel nicht mit realen Materialien gearbeitet, die verbraucht werden, abgesehen von der Arbeitskraft selbst. Das klingt zunächst vielleicht trivial, ist aber ein entscheidender Unterschied. Die langwierigen Prozesse im Gebäudebau existieren nicht zuletzt deswegen, weil man sich hier schon aus Gründen des Materialsverlusts keine großen Fehler erlauben darf. Ist das Brett gesägt, die Mauer errichtet oder das Fundament gegossen, dann kann man nicht mehr so leicht zurück, denn nicht nur kostet das viel Zeit, auch muss neues Material besorgt werden, denn das alte ist nun verloren.

In der Softwareentwicklung ist das anders. Softwareentwickler verbrauchen kein Material. Ihr Material ist virtuell, lässt sich bei Bedarf verlustfrei duplizieren, teilen, neu zusammensetzen – und das so oft man will. Das ist wohl auch einer der Reize, Software zu bauen. Man kann komplexe Systeme bauen ohne großen Materialaufwand. Es sind vielleicht diese Freiheit und die Entkopplung vom Materialaufwand, die manchen Softwareentwickler zuweilen denken lässt, dass es keinen Grund gäbe, langwierige Planungen vorzunehmen, wie es eben z. B. beim Gebäudebau notwendig ist. Klar, die Bauingenieure brauchen Bau- und Zeitpläne, weil sonst ein schiefes Gebäude entsteht und vielleicht einstürzt. Aber bei Software kann man ja jederzeit, wenn was nicht passt, die Entfernen-Taste drücken und eine verbesserte Version einsetzen. Man ist losgelöst von der Gefahr, Material zu vergeuden und nicht mehr zurückzukönnen. Warum also planen, warum nicht einfach anfangen und dann zuschauen, wie sich das virtuelle Konstrukt langsam entwickelt, und einfach gegensteuern, wenn etwas nicht korrekt funktioniert?

Der Gedankengang hat einen Haken. Er nimmt an, wir hätten beliebig viel Zeit zur Verfügung und dass wir innerhalb unserer Zeit über Versuch und Irrtum zum Ziel kommen und dabei alle unsere Ziele erreichen könnten. Das ist natürlich nicht so. Jeder Entwickler weiß, dass Softwareprojekte sich in zumeist enorm engen Zeitplänen bewegen. Bei unsauber geplanten Projekten führt dies oft dazu, dass vielleicht die Anwendung noch grundsätzlich gerade so funktioniert, dass sie aber meistens nicht fehlerfrei ist und dass Nebenziele wie Wartbarkeit und Lesbarkeit selten zufriedenstellend erreicht werden.

Damit erklärt sich auch, dass Softwareprojekte natürlich einen Materialeinsatz haben, nämlich genau die erwähnte Zeit. Wenn ein Flash-Entwickler eine Komponente baut, von der man hinterher feststellt, dass man sie so gar nicht gebrauchen kann, dann ist die Komponente das Resultat von Materialverschwendung, nämlich von Zeit. In einem kleinen Projekt mag es vielleicht nicht so auffallen, wenn mal etwas unnötigerweise programmiert wurde, in großen Projekten, wo vielleicht aufgrund schlechter Planung ein ganzes Modul nicht verwendet werden kann, weil es nicht den Anforderungen genügt, wirkt sich das teilweise erheblich auf die Gesamtkosten und auf das Gesamtprojekt aus, weil oft die anderen Module von diesem in irgendeiner Art und Weise abhängen und meist der Zeitplan nicht mehr gehalten werden kann.

Auch stimmt die Annahme nicht, es gebe im Bereich der Softwareentwicklung nicht so was wie Fundamente, die – einmal gegossen – nicht mehr verändert werden können. Auch in Softwareprojekten ist irgendwann ein Zeitpunkt erreicht, wo man zentrale Teile einer Anwendung nicht mehr ohne Weiteres ändern kann, weil viele andere Teile stark davon abhängen. Nun könnte man natürlich argumentieren, dass die Anwendung in diesem Fall schlecht strukturiert wäre, aber auch in einer gut strukturierten Anwendung gibt es Teile, die zentrale Aufgaben übernehmen und z. B. stark mit den zuvor definierten Geschäftsprozessen verknüpft sind. Man kann sie dann zwar auch noch ändern, aber der Aufwand wird erheblich sein.

Wichtig ist auch, dass Software zwar virtuell ist, sie aber nie in einer virtuellen Welt eingebettet ist, sondern in einer realen Welt. Das bedeutet, dass sich die Entwicklung eines Programms nicht nur den Zwängen und Limitierungen der Programmiersprache oder der virtuellen Maschine zu unterwerfen hat, sondern auch den realen Sachzwängen der sie umgebenden Welt. Wenn also z. B. ein Auftraggeber durch Nutzertracking herausgefunden hat, dass der überwiegende Teil seiner Zielgruppe den Flash Player einer ganz bestimmten Version verwendet, dann hilft alles nichts, dann muss für diesen Player entwickelt werden, auch wenn dies bedeutet, dass bestimmte Features eventuell nicht genutzt werden können. Wenn ein Aufraggeber intern bei all seinen Projekten eine bestimmte Serverkomponente einsetzt und er aus Kosten- oder auch Know-how-Gründen auch beim neuen Projekt diese Komponente wieder einsetzen will, dann muss das mit den restlichen Anforderungen abgewägt werden.

Eine Struktur zu entwickeln, die sowohl die konkreten funktionalen Aspekte einer Aufgabenstellung als auch die Sachzwänge des Projekts berücksichtigt, ist die schwierige, aber auch spannende Aufgabe jedes Softwareentwicklers, im Großen wie im Kleinen. Der Softwareentwurf erlaubt uns hier, Ideen zu entwickeln und in der Theorie auszuprobieren, ohne dass wir bereits große Mengen an Code schreiben müssen. Während des Entwurfs können wir Fehler machen. Da es sich quasi um Trockenübungen handelt, kosten diese Fehler weniger, als wenn wir alles gleich durchprogrammieren würden. Wir können mit Strukturen spielen und ausprobieren, ob sie der Belastung der vielen Anforderungen standhalten würden.

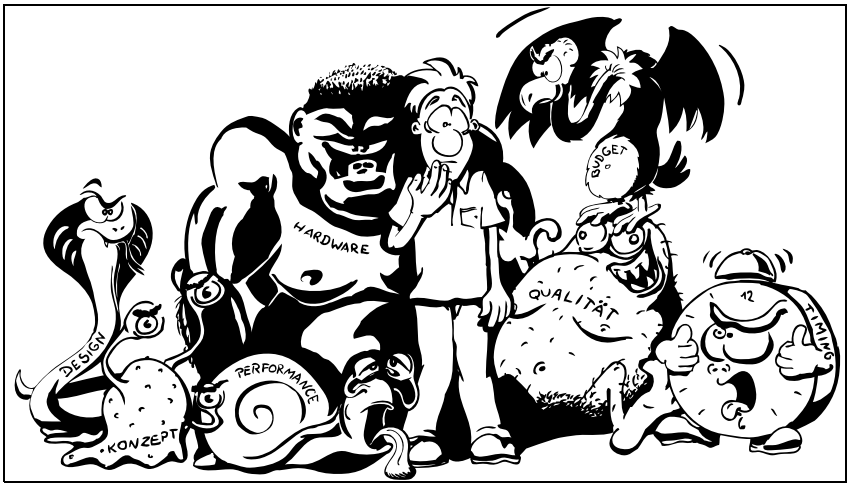


Abbildung 3.2: Viele unterschiedliche Anforderungen müssen in einem Softwareprojekt berücksichtigt werden.

Das Besondere am Softwareentwurf ist nun, dass es keinen allgemeingültigen Weg zu einer gut strukturierten Anwendung gibt, denn die meisten Anwendungen, die heutzutage entwickelt werden, sind Individualanwendungen, deren Problemkreis sich so gut wie immer von vorangegangenen Projekten unterscheidet. Deswegen ist der technische Entwurf einer Anwendung auch ein kreativer Prozess, denn es geht hier nicht um das Reproduzieren von bekannten Mustern, sondern meistens um das erstmalige Finden einer Struktur, die für das anstehende Problem am besten geeignet scheint. Dass sich in diesen Strukturen dann im Detail doch oft wieder Muster finden (von denen ich in Kapitel 4 einige beschreiben werde), ist tröstlich, ändert aber nichts an der Herausforderung.

Wie man nun beim Entwurf genau vorgehen sollte, wird in Kapitel 6 organisatorisch und in diesem Kapitel inhaltlich erklärt. Nicht immer kann oder sollte man die komplette Struktur einer Anwendung von Anfang an erarbeiten, geschweige denn im Kopf haben. Der Entwurf sollte nicht als eine lästige Aufgabe angesehen werden, die man erst erledigen muss, bevor man umsetzen darf. Der Entwurf sollte vielmehr als ein Hilfsmittel angesehen werden, den wir im Wechsel mit der Umsetzung immer wieder einsetzen können, bevor wir den nächsten Schritt tun. Wenn das nächste Modul oder der nächste Teilbereich der Anwendung angegangen werden soll, dann entwerfen wir zuerst ein paar Ideen, wie wir diesen Teil strukturieren könnten, prüfen die Idee auf ihre Tragfestigkeit und setzen sie dann um. So kommen wir Schritt für Schritt zur Gesamtanwendung.

Die Entwicklungen im Bereich des Software Engineerings der letzten Jahrzehnte haben auf eines kontinuierlich hingewirkt: die Verringerung von Komplexität beim Bau von Software. Heutige Anwendungen, auch im Flash-Bereich, sind bereits so komplex, dass nur noch selten ein einzelner Flash-Entwickler – sehr kleine Projekte vielleicht ausgenommen – die komplette Anwendung bis in Details überblickt. Und auch bei einem kleinen Projekt, das eventuell Drittbibliotheken verwendet, wird ein Flash-Entwickler eher selten die genaue

Arbeitsweise dieser Bibliotheken kennen. Durch den Einsatz dieser Bibliotheken hat dieser Entwickler aber schon eine wichtige Entscheidung getroffen: Er hat für sich Komplexität reduziert, denn er muss sich nicht für den internen Aufbau der Bibliothek interessieren, er muss lediglich wissen, wie sie funktioniert. Die Entscheidung, in einem Projekt eine Bibliothek einzusetzen, ist eine der vielen Grundinstrumentarien, die Softwareentwickler zur Verfügung haben, um Komplexität zu verringern, neben vielen anderen, die wir noch besprechen werden. Es ist eine Entscheidung im Bereich der Softwarearchitektur.

3.1 Architektur versus Entwurf

Im Software Engineering schwirren die Begriffe Architektur und Entwurf bzw. Design mit unterschiedlichen Bedeutungen herum. Wenn manch ein Entwickler sagt, die Architektur seiner Anwendung sei den Anforderungen gerecht geworden, dann ist man nicht immer sicher, ob er die konkrete Klassenstruktur meint oder allgemeine Entscheidungen, beispielsweise dass er Cairngorm eingesetzt hat. Es gibt aber einen Unterschied zwischen der Architektur eines Softwaresystems und dem Entwurf bzw. Design (mit Design ist hier immer der Softwareentwurf gemeint, nicht etwa die visuelle Gestaltung), und es ist auch sinnvoll, diese Unterscheidung zu treffen.

Kazman, Clements und Bass schreiben:

»The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.« (Bass et al. 2008, S. 21)

Man kann also grob sagen: Eine Softwarearchitektur beschreibt den Aufbau und das Zusammenspiel einzelner Softwareelemente. Eine Softwarearchitektur kann dabei eingebunden sein in eine Systemarchitektur, bei der dann auch noch neben der Software die Hardware eine Rolle spielt. Eine Systemarchitektur würde also konkret auch die Hardwarekomponenten und ihre Verknüpfungen beschreiben.

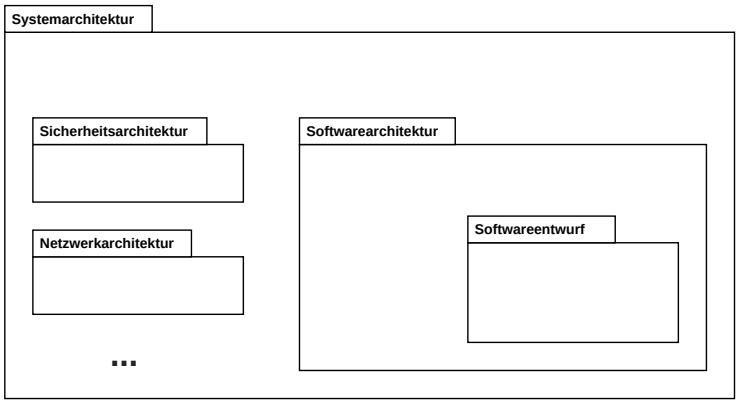


Abbildung 3.3: Einordnung von System- und Softwarearchitektur sowie Softwareentwurf

Der Softwareentwurf hingegen beschreibt jeweils den inneren Aufbau und die Funktionsweise einer einzelnen Softwarekomponente, also zum Beispiel eines konkreten Flash-Moduls.

3.1.1 Softwarearchitektur

In der Softwarearchitektur werden globale Entscheidungen getroffen, die die Anwendung als Ganzes betreffen. Z. B. wird hier die Entscheidung getroffen, ob und, wenn ja, welche Flash Player-Version verwendet wird. Alle Softwarekomponenten werden festgelegt, z. B. welcher Application Server kommt zum Einsatz, welcher Webserver, wird Clustering verwendet usw. Außerdem werden die grundsätzlichen Schnittstellen festgelegt, z. B. über welches Schnittstellenprotokoll spricht Flash mit dem Server (z. B. Remoting, XML etc.). Es werden prinzipielle Vorgaben für den folgenden Softwareentwurf gemacht, z. B. sollen objektorientierte Prinzipien eingesetzt werden oder vielleicht Aspektorientierung. Es werden grundsätzliche Frameworks vorgegeben, also Flex, Einsatz von Architektur-Frameworks wie Cairngorm usw. Auch Anforderungen, die Auswirkungen auf spätere Phasen des Projekts haben können, werden hier definiert, wie dass alle Teile der Anwendung Nutzertracking implementieren sollen und deswegen dafür ein zentrales Modul vorgesehen werden muss.

Die Softwarearchitektur schaut bei einem Projekt also auf einen recht hohen Abstraktionslevel. Hier geht es noch nicht um einzelne Module oder gar um Klassen. Hier wird ermittelt, welche grundsätzlichen Teile für die Anwendung notwendig sind und wie diese zusammenwirken sollen. Dabei muss der Softwarearchitekt zusammen mit dem Kunden ausloten, welche Systeme eingesetzt werden können, welche Limitierungen es gibt und welche konkreten Anforderungen an die Anwendung gestellt werden, die eventuell Einfluss auf die Gesamtkonstruktion der Anwendung haben können. In den meisten Fällen bringt der Kunde eine ganze Reihe an Einschränkungen und Anforderungen mit, die in der Softwarearchitektur zu berücksichtigen sind. Dazu kann gehören, dass das Hosting einer Website auf betriebsinternen Servern durchgeführt werden soll, die eine bestimmte Ausstattung haben, die nicht oder nur eingeschränkt verändert werden kann. Das kann Auswirkungen auf die dort installierbaren Komponenten haben, wie z. B. Typ und Version des Webservers oder Application Servers. Die eingesetzte Infrastruktur kann es eventuell nötig machen, dass möglichst wenig serverseitig dynamische Dienste angeboten werden sollen, weil die Belastungsfähigkeit der Server eingeschränkt ist. Eventuell kann nur eine bestimmte Version des Flash Players eingesetzt werden, weil nur diese auf den Rechnern des Kunden verfügbar ist und sie nicht so schnell unternehmensweit geupdatet werden kann.

Der Softwarearchitekt muss zusammen mit dem Kunden alle Bedingungen und Anforderungen identifizieren, die einen Einfluss auf die geplante Anwendung haben können, und gerade bei internetbasierten Systemen ist meist eine Menge kleiner Teilbereiche involviert.

Auch Anforderungen, die sich aus der Anwendung selbst ergeben, können einen starken Einfluss auf die Architektur haben. Wenn die Anwendung eine sehr hohe Ausfallsicherheit haben muss, weil von ihr z. B. andere Unternehmensprozesse abhängen (z. B. ein Online-Shop bei einem Unternehmen, was seine Produkte hauptsächlich online vertreibt), dann

muss in der Softwarearchitektur entsprechende Redundanz vorgesehen werden, also das mehrfache Vorhandensein einer Komponente, falls eine von ihnen mal ausfällt.

Eine Anwendung, die eine sehr breite Zielgruppe ansprechen soll, kann wiederum eventuell nicht nur auf Flash als Client-Technologie setzen, sondern muss Alternativen für Nutzer ohne Flash anbieten.

Eine Anwendung, die eine hohe zu erwartende Lebensdauer hat, muss von der Architektur her eventuell modularer aufgebaut sein, damit mit der Zeit veraltete Teile gegen neue ausgetauscht oder damit fehlerhafte Teile isoliert vom Rest der Anwendung verbessert werden können.

Die Softwarearchitektur bestimmt also das grundsätzliche Gerüst der Anwendung. Fehlentscheidungen in der Architektur können enorm negative Auswirkungen in allen nachfolgenden Phasen eines Projekts zur Folge haben, die teilweise nur durch gewaltige Aufwände wieder ausgebügelt werden können. Deswegen ist es umso wichtiger, dass die grundsätzliche Architektur einer Anwendung sorgfältig durchdacht wird. Iterative Vorgehensweisen wie bei der agilen Softwareentwicklung versuchen, den Druck vom Bereich der Softwarearchitektur zu nehmen, indem sie in wiederkehrenden Zyklen kleinere Teile einer großen Anwendung beschreiben. Nichtsdestotrotz müssen bestimmte Grundentscheidungen bereits zu Beginn getroffen werden, was bedeutet, dass sich Kunde und Dienstleister auf bestimmte Eckpfeiler bereits zu Beginn eines Projekts einigen müssen.

Ich werde in diesem Buch das Thema Softwarearchitektur nicht im Detail behandeln, weil es aus Sicht von Flash-/Flex- und ActionScript-Entwicklung ein vorgelagertes Thema ist und ich die Themen rund um den Softwareentwurf für Flash- und Flex-Entwickler für wichtiger erachte.

3.1.2 Softwareentwurf

Beim Softwareentwurf geht man nun wieder eine Ebene tiefer. Wurden in der Architektur die einzelnen Komponenten und ihre grundsätzlichen Schnittstellen erfasst, so werden diese einzelnen Komponenten nun genauer unter die Lupe genommen. Wie schon im vorigen Abschnitt erwähnt, besteht ein Gesamtsoftwaresystem zumeist aus Modulen unterschiedlichster Art.

Im Softwareentwurf beschreiben wir diese Module nun im Detail. Dabei orientiert man sich wieder zunächst an den Anwendungsfällen und den fachlichen Klassen und Objekten, die wir in Kapitel 2 kennengelernt haben, und schaut nun, wie man diese technisch so strukturieren kann, dass daraus später eine Anwendung gebaut werden kann. Die Schwierigkeit besteht hier nun einmal darin, dass eine Anwendung nicht nur eine Ansammlung fachlicher Klassen ist, sondern auch aus sehr technischen Objekten besteht, die mit dem fachlichen Problem erst einmal nichts zu tun haben. Eine Preloader-Sequenz z. B. ist ein rein technisches Konstrukt und findet sich natürlich nicht in der fachlichen Beschreibung der Anwendungsfälle wieder. Beim Softwareentwurf müssen wir also zusätzlich zu den fachlichen Klassen und Objekten auch noch technische Objekte hinzufügen, die letztlich die Anwendung komplettieren. Auch die Struktur, die sich dabei herauskristallisiert, ist nun

eine Mischung aus den Beziehungen, die wir schon in der fachlichen Beschreibung modelliert hatten, und den technischen Schnittstellen, die sich erst jetzt ergeben.

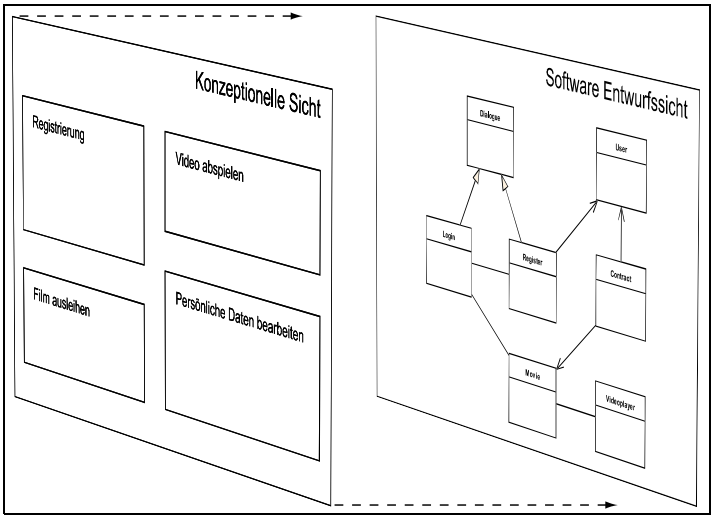


Abbildung 3.4: Aus der konzeptionellen Sicht wird der Entwurf abgeleitet.

Eine wichtige Frage, die sich beim Softwareentwurf immer stellt, ist die Frage, wie tief man einsteigen will. In Kapitel 6 werde ich noch darauf zu sprechen kommen. Die iterativen Vorgehensmodelle legen nahe, dass man zu Beginn nicht die gesamte Anwendung bis ins Detail entwerfen sollte bzw. kann. Softwareprojekte sind häufig von mehreren Änderungen oder Erweiterungen der Anforderungen gekennzeichnet, sodass ein kompletter Entwurf der Anwendung nicht lange Bestand haben würde. Die Grundregel lautet also, man entwirft schrittweise einzelne, überschaubare Teile – und das iterativ immer wieder.

Zusätzlich ist es unverzichtbar, für die Gesamtanwendung zumindest eine grundsätzliche Struktur zu entwerfen, die die Basis für die folgenden Iterationen bildet. Der Knackpunkt hier ist, dass der Fokus der Entwickler die meiste Zeit auf Details liegt anstatt auf der Gesamtanwendung. Pro Iteration müssen einzelne Module fertiggestellt und integriert werden, dabei betrachtet man die Module und ihre unmittelbar benachbarten Module, aber nicht zwingend die Gesamtstruktur. Die Gefahr ist, dass man zu spät die Gesamtstruktur begutachtet, um noch eingreifen zu können. Ohne eine Grundstruktur, ein Gerüst, besteht die Gefahr, dass sich das Projekt verselbstständigt und in einer Gesamtstruktur mündet, die irgendwann einem Flickenteppich gleicht und nur noch schwer zu warten ist.

Deswegen ist es wichtig, dass bereits zu Beginn eine allgemeine Struktur erarbeitet wird, die den Rahmen für die kommende Entwicklung vorgibt. Innerhalb einer Iteration kann man sich dann detailliert an einen Aspekt der Anwendung machen und diesen entwerfen. Wie detailliert man dabei vorgeht, hängt von vielen Faktoren ab. Ist die Person, die den Entwurf macht, die gleiche, die auch die Umsetzung machen wird? Wenn nein, ist die Person, die die Umsetzung macht, erfahren oder noch ein Berufseinsteiger? Was sind die Anforde-

rungen an die Dokumentation der Anwendung? Wird eine detaillierte Beschreibung des Entwurfs vielleicht für kommende Module noch benötigt? Unabhängig von diesen Fragen wird man während des Softwareentwurfs selten tiefer als bis zur Definition der öffentlichen Methoden und Eigenschaften einer Klasse gehen, die also die Schnittstelle definieren, denn alle Belange, die noch tiefer gehen, werden normalerweise erst während der konkreten Implementierung entschieden.

Objektorientierter Entwurf

Objektorientierter Entwurf ist nach der objektorientierten Analyse der nächste Schritt hin zu einem Softwareprodukt. Wobei nächster Schritt, wie in diesem Buch schon oft erwähnt, nicht wörtlich zu verstehen ist, weil je nach Vorgehensweise öfter zwischen Anforderungsanalyse und Softwareentwurf sowie Implementierung hin und her gesprungen wird.

Wurde in der Analyse das Problem eingekreist und beschrieben, so versuchen wir beim Entwurf nun, die Lösung zu skizzieren. In der Analyse haben wir das Problemfeld – z. B. den Online-Videoverleih in unserem Beispiel der Firma FilmRegal – analysiert, Geschäftsfälle identifiziert, Anwendungsfälle beschrieben und fachliche Klassen modelliert. Wir haben die vor uns stehende Aufgabe fachlich erfasst. Nun geht es an die Beschreibung der Lösungsidee, also der Idee, die die Umsetzung der fachlichen Aufgabe mittels Software beschreibt.

Weil wir objektorientiert arbeiten, gelten auch beim Entwurf wieder ähnliche Methodiken wie bei der objektorientierten Analyse. Am wichtigsten jedoch ist, wir verwenden die Ergebnisse, die Modelle und Beschreibungen aus der Analysephase als Basis für unsere Modelle im Entwurf. Haben wir also ein fachliches Objektmodell in der Analyse- bzw. Konzeptionsphase erstellt, so können wir dies nun nahtlos als Basis für unsere Überlegungen in der Entwurfsphase weiterverwenden.

Es kommt nicht selten vor, dass z. B. viele der fachlichen Klassen zu konkreten Klassen in der Anwendung werden. Wurden im Fachkonzept fachliche Klassen wie Film, Leihvertrag und Kunde definiert, so kommen solche Klassen dann oft direkt auch in der Anwendung vor, natürlich viel konkreter.

3.2 Komplexität

Die ersten Computerprogramme Mitte der Vierzigerjahre des letzten Jahrhunderts waren hauptsächlich für die Mathematik bestimmt. Sie wurden auf Lochkarten notiert. Die Komplexität dieser Programme hielt sich in Grenzen, wenn man sie mit heutigen Programmen vergleicht. Mit Weiterentwicklung der Hardware, also mit Zunahme der Rechenkapazität, und vor allem mit Erfindung neuer Schnittstellen zu anderen Systemen wie z. B. Druckern, Maschinen usw. ergaben sich immer mehr Einsatzbereiche, und die Computerprogramme mussten nun nicht mehr nur mathematische Berechnungen durchführen, sondern ungleich mehr. Damit stieg natürlich auch unweigerlich die Komplexität der Computerprogramme. Je mehr Möglichkeiten sich ergaben, Probleme und Aufgaben mit dem Computer zu lösen, umso komplexer wurden auch die Anforderungen an diese Programme.

Die Flash-Entwickler unter Ihnen, die die Geschichte von Flash von relativ früh an mitgemacht haben, können das bestimmt auch nachvollziehen. Ich selbst habe mit Flash 3 begonnen. Die Möglichkeiten von Flash 3 sind mit denen der heutigen Flash-Plattform nicht zu vergleichen. Die Komplexität der Animationen und kleinen Anwendungen, die man zu der Zeit mit Flash umsetzen konnte, war im gleichen Maße überschaubar. Je mehr Flash und ActionScript an Funktionalität wuchsen, je performanter der Flash Player mit der Zeit wurde, umso komplexer wurden auch die Anwendungen, die damit umgesetzt wurden.

Bestanden frühe Präsentationen noch aus relativ wenig Code, der meist noch direkt in der Flash IDE direkt eingegeben wurde, so konnte man mit Flash 5 schon recht komplexe Anwendungen mit ausgelagertem Code bauen. Einzig die Performance hielt hier die Komplexität eventuell noch im Zaum, und manche Arten von Anwendungen waren noch nicht möglich, weil die Flash API schlichtweg nicht die entsprechenden Funktionen bereithielt.

Heute ist Flash zu einer recht großen Plattform herangewachsen und bietet eine Fülle von Funktionalitäten, die für einen einzelnen Entwickler nur noch schwierig bis gar nicht komplett zu beherrschen ist. Und im gleichen Maße steigt auch die Komplexität der mit Flash umgesetzten Anwendungen, die man im Internet bewundern kann, an. Es ist also nicht nur die Komplexität der zur Verfügung stehenden Funktionen der Flash API gemeint, sondern auch die Komplexität der Anwendungen, die mit Flash und Flex umgesetzt werden können.

Was aber ist das Problem bei der Komplexität? Menschen haben Schwierigkeiten mit komplexen Vorgängen oder Zusammenhängen. Wir können zwar viele Informationen abspeichern, aber nicht in kurzen Zeiträumen. Wir erinnern uns zwar an viele Dinge in unserem Leben, aber wir können uns bewusst nur mit einem kleinen Teil an Informationen auseinandersetzen. Deswegen zerlegen wir gedanklich komplexe Strukturen so lange, bis die kleineren Teile wieder von uns erfassbar sind. Besonders schwer fallen uns komplexe Systeme, die sich über den Faktor Zeit erstrecken. Ein simples Beispiel hier ist die Heizung. Wenn wir ein Thermometer in die Hand bekommen und in einem Raum den Thermostaten an einem Heizkörper so einstellen sollen, dass eine bestimmte Temperatur im Raum erreicht wird, dann nähern wir uns dieser Temperatur meist nur mit großen Schwankungen an, weil ein Drehen am Thermostat ja erst nach einiger Zeit seine Wirkung zeigt. Dietrich Dörner schreibt:

»Die Existenz von vielen, voneinander abhängigen Merkmalen in einem Ausschnitt der Realität wollen wir als »Komplexität« bezeichnen.« (Dörner 2008, S. 60)

Damit ist gemeint, je mehr Eigenschaften sich in einer Problemstellung befinden, die sich gegenseitig beeinflussen oder voneinander abhängen, als desto komplexer empfinden wir die Problemstellung. Dabei ist nicht nur die Anzahl der Eigenschaften von Belang, sondern ihre Vernetztheit. Es ist schwer, ein System zu kontrollieren, das aus vielen Merkmalen besteht, die sich gegenseitig beeinflussen. Software kann zu so einem System werden. In einem Projekt mit Entwicklern, Designern, Konzeptern und anderen Teammitgliedern, in dem Module gebaut werden mit vielen Klassen, Dateien und Schnittstellen, wird schnell ein Level an Komplexität erreicht, den ein einzelner Entwickler kaum noch überschauen kann.

Schon allein die Komplexität der Flash API an sich führt z. B. seit einigen Jahren zu einer Spezialisierung von Kompetenzen unter den Entwicklern. Gerade mit dem Aufkommen von Themen wie 3D, Sound, Grafik, Netzwerkprogrammierung und dergleichen hat die Flash-Welt schon für sich eine Komplexität angenommen, die ein einzelner Entwickler nicht mehr allein in all seinen Facetten komplett beherrschen kann. Sicherlich kann man gewisse Grundkenntnisse in all diesen Themen haben, und ich will auch nicht ausschließen, dass es einige Supertalente gibt, die tatsächlich fast alle Bereiche sehr gut abdecken, aber der durchschnittliche Flash-/Flex-Entwickler ist wohl eher froh, wenn er mit den wichtigsten Neuerungen mithalten kann und sich dann bei Bedarf in Themen einarbeitet, wenn es nötig ist. So ergibt es sich automatisch, dass einige unter uns eher Detailwissen im 3D-Bereich entwickelt haben, andere haben sich ganz und gar der Soundprogrammierung verschrieben, wieder andere experimentieren mit der Verknüpfung von Flash zu anderen Systemen und Programmumgebungen usw.

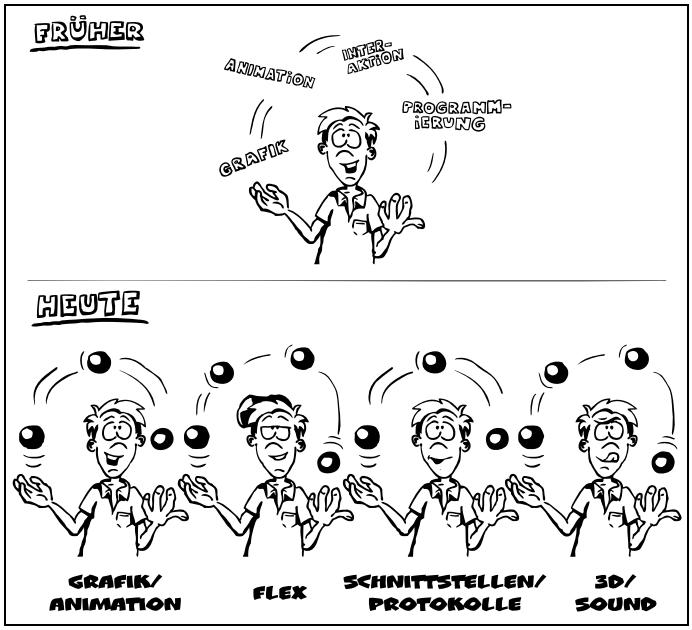


Abbildung 3.5: Kompetenzverteilung von Flash-Entwicklern, früher und heute

Wenn es aber also so viele Spezialgebiete und sicherlich auch Anwendungen gibt, die einen Teil dieser Gebiete in sich vereinen, dann entsteht bereits dadurch Komplexität, all diese Fachgebiete unter eine gemeinsame Kontrolle zu bekommen und bestmöglich gemeinsam einzusetzen.

Ein weiterer Faktor für Komplexität sind die Anwendungen. Aufgrund der weitreichenden Möglichkeiten der Flash API, der vielen Schnittstellen, die sie anbietet, der hohen Performance des Flash Players und der mittlerweile recht ausgereiften Programmiersprache ActionScript ist der Einsatzbereich für Anwendungen kaum begrenzt. Grundsätzlich ergibt

sich natürlich eine Begrenzung darin, dass mit Flash/Flex hauptsächlich Client-Anwendungen erstellt werden, denn Flash ist eine Client-Technologie, aber innerhalb dieser Domäne gibt es prinzipiell kaum Einschränkungen, außer solche, die sich durch die eingesetzte Hardware und daraus resultierende Limitierungen hinsichtlich Performance, Speicher usw. ergeben.

Die hierdurch entstehende Komplexität hat einmal eine inhaltliche Komponente. Die Anwendungen, die entstehen, entspringen ja immer aus einem fachlichen Ursprung. Anwendungen kommen aus dem Bereich des Marketings, des Finanzwesens, Geschäftswesens allgemein, der Medizin, Wissenschaft, des Tourismus, der Unterhaltung, Industrie und viele, viele mehr. Jeder dieser Bereiche hat schon seine eigene fachliche Komplexität. Wenn wir im Bereich der Medizin eine Anwendung entwickeln sollen, dann müssen wir als Entwickler in einem gewissen Maß auch die fachlichen Zusammenhänge verstehen, die in diesem Bereich eine Rolle spielen, wie sollten wir sonst eine sinnvolle Anwendung für diesen Bereich schreiben.

Sich dieses Wissen anzueignen, ist also unter Umständen bereits eine enorm herausfordernde Aufgabe an sich. Nicht umsonst kann man heutzutage in vielen Stellenangeboten für Softwareentwickler lesen, dass eine gewisse Expertise in einem fachlichen Segment erwünscht ist. Und nicht zuletzt kann man an den Hochschulen Wirtschafts-, medizinische und Medieninformatik studieren, weil es eben sehr unterschiedliche fachliche Anforderungen für Softwareentwickler in diesen jeweiligen Bereichen gibt.

Eine weitere Komponente der Komplexität liegt dann noch in der Größe der Anwendungen, die erstellt werden. Flash- und Flex-Anwendungen können im Prinzip eine beliebige Größe annehmen. Über Nachlademechanismen ist es grundsätzlich möglich, enorm komplexe, modulare Applikationen zu bauen. In der schieren Größe solcher Anwendungen liegt natürlich selbst auch eine Art von Komplexität. Und mit dieser Größe ergeben sich wiederum Anforderungen, die die Komplexität erhöhen. Ein großes Projekt kann nicht von einem Entwickler allein umgesetzt werden. Ein Team von vielen Entwicklern wiederum birgt für sich schon neue Arten von Komplexität, die sich in Form von Kommunikation und der Organisation der Zusammenarbeit zeigen.

Viele Flash-Entwickler wissen bereits aus Erfahrung, dass sich große Projekte nicht einfach durch viele Programmierer lösen lassen. Denn je mehr Entwickler an einem Projekt beteiligt sind, umso höher ist der Koordinationsaufwand, sprich die Komplexität der Organisation. Und irgendwann ist dann der Punkt erreicht, wo die vielen Entwickler nicht mehr effektiv zusammenarbeiten können, weil sich die Aufgaben z. B. nicht mehr weiter aufteilen lassen oder weil der Aufwand, die Aufgaben abzustimmen, zu hoch ist.

Wir sehen also, Komplexität steckt in vielen Ecken und Winkeln unserer Arbeit als Entwickler. Wir haben aber nun mal nur eine begrenzte Kapazität, Informationen zu verarbeiten. Um also nicht handlungsunfähig zu werden, weil wir mit zu vielen Dingen gleichzeitig fertig werden müssen, müssen wir für eine einzelne Person die Komplexität zu jedem Zeitpunkt im Zaum halten. Steven McConnell sagt: »Ich vertrete [...] die Ansicht, dass das erste Gebot bei der Softwareentwicklung lauten muss: Halte die Komplexität im Griff.« (McConnell 2007, S. 80)

Wir müssen deswegen in all diesen Bereichen versuchen, die Komplexität so weit zu verringern, bis wir unsere Aufgabe erfüllen zu können. Wie aber kann man die Komplexität verringern? Man kann ja schließlich nicht einfach bestimmte Faktoren außer Acht lassen. Müssen nicht alle oben beschriebenen Bereiche berücksichtigt werden? Die Antwort lautet: Ja, aber nicht zur gleichen Zeit und auch nicht zwingend von allen Projektbeteiligten gleichermaßen. Der Schlüssel liegt darin, die Fülle an Aufgaben und Informationen zum einen zeitlich und zum anderen personell zu verteilen. Das Ziel ist also, dass sich ein Entwickler zu einem bestimmten Zeitpunkt nur mit einer Menge an Informationen und Aufgaben beschäftigen muss, die er auch verarbeiten kann, ohne den Überblick über seinen Verantwortungsbereich zu verlieren. Hat er diese Aufgaben erledigt, kann er sich wieder mit einem anderen Bereich beschäftigen, diesen abarbeiten usw.

Um also der Komplexität Herr zu werden, müssen wir das Projekt in so handliche Pakete aufteilen, die pro Entwickler einzeln in einer annehmbaren Zeit bearbeitbar sind. Wir werden in Kapitel 6 noch sehen, dass dies auf ganz unterschiedlichste Art gelöst wird. Generell ist bei vielen Vorgehensweisen erkennbar, dass man sich vom Allgemeinen ins Spezielle vorarbeitet, Top-down auf Englisch. Wir haben schon gesehen, dass z. B. das Gesamtprojekt zunächst einmal in grobe Arbeitsabschnitte aufgeteilt wird, z. B. in die Analysephase, die ihrerseits in kleinere Abschnitte unterteilt ist, wie Definition der Geschäftsprozesse, Beschreibung der Anwendungsfälle, Definition der Fachklassen usw. Hier ist klar eine Richtung vom Allgemeinen (Geschäftsprozess) zum immer Spezielleren (fachliche Klassen) erkennbar.

Sie mögen sich vielleicht schon gefragt haben, warum denn so viele Arbeitsschritte notwendig sind für die Anforderungsanalyse. Wie schon gesagt, bei kleinen Projekten wird man sicherlich einige Schritte zusammenfassen, aber bei größeren Projekten bedeuten die einzelnen Arbeitspakete auch eine Möglichkeit zur zeitlichen und personellen Aufteilung und also letztlich, dass wir Komplexität für den einzelnen Entwickler reduzieren können.

Auch im Softwareentwurf stehen wir wieder vor dem gleichen Problem. Und auch hier müssen wir wieder über die zeitliche oder personelle Aufteilung versuchen, Komplexität zu reduzieren. Die Objektorientierung wird uns auch hier wieder helfen, denn mittels des objektorientierten Entwurfs können wir unsere Anwendung sehr elegant in Komponenten, Module, Pakete, Klassen, Interfaces und Objekte aufteilen, die natürlich nicht unbedingt nur den Zweck haben, die Komplexität für den einzelnen Entwickler zu reduzieren, aber neben anderen Zielen eben auch das.

Damit diese zeitliche und personelle Aufteilung gelingt, sollten die einzelnen Aufgaben oder Arbeitspakete idealerweise unabhängig voneinander sein. Das gilt nicht nur für Module in unserer Anwendung, sondern auch für konkrete Arbeitsschritte. Ideal wäre natürlich, wenn alle Arbeitspakete keinerlei Abhängigkeiten von anderen hätten, denn dann könnte man sie komplett frei planen. Das ist natürlich nicht so. Zwischen der Anforderungsanalyse und dem Softwareentwurf gibt es natürlich grundsätzlich die Abhängigkeit, dass der Softwareentwurf nur die Themen schon bearbeiten kann, die in der Anforderungsanalyse schon definiert wurden oder die sich aus allgemeinen, in der Anforderungsanalyse definierten Rahmenbedingungen ergeben. Es gibt also sequenzielle Abhängigkeiten, die wir nicht durchbrechen können. Abgesehen von diesen aber sollten innerhalb

von einem Arbeitsschritt, also zum Beispiel dem Softwareentwurf, möglichst wenige Querabhängigkeiten existieren. Wenn also Entwickler A ein Flash-Modul entwirft, sollte diese Tätigkeit möglichst wenige Abhängigkeiten zu dem Entwurf eines anderen Flash-Moduls haben, das Entwickler B entwirft, abgesehen von allgemeinen Schnittstellen, die beide Module eventuell miteinander haben. Denn je weniger die beiden Entwickler sich miteinander abstimmen müssen, umso mehr können sie sich auf ihre eigenen Module konzentrieren. Um das zu erreichen, müssen die Module selbst möglichst unabhängig voneinander sein und die eventuell vorhandenen Schnittstellen so simpel wie möglich. Aber dazu kommen wir später.

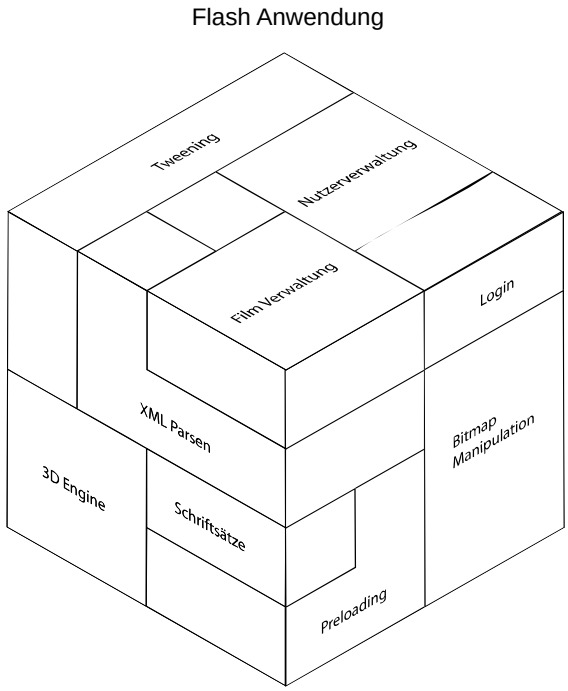


Abbildung 3.6: Die Komplexität einer Anwendung kann in Einzelteile zerlegt werden.

3.3 Projektziele

Was sind die Ziele eines Flash-Entwicklers innerhalb eines Projekts? Das primäre Ziel ist natürlich, dass die Anwendung die in den Anforderungen definierten Aufgaben erfüllt unter Beachtung aller Randbedingungen – und das innerhalb des Zeitplans und unter Einhaltung des zur Verfügung stehenden Budgets. Das allein ist schon schwer genug, und in vielen Projekten wird nicht mal dieses Ziel komplett erreicht. Projekte werden zu spät fertig, oder manche Funktionalität muss zurückgestellt werden, damit der Zeitplan eingehalten werden kann, Budgets werden überschritten.

Aber neben diesem primären Ziel gibt es immer auch noch sekundäre Ziele, die für ein Projekt wichtig sein können. Nicht immer sind alle von gleicher Bedeutung, aber zu einem gewissen Anteil spielen sie alle eine Rolle. Die klassischen, zugegebenermaßen sehr allgemein formulierten sekundären Ziele sind: Wartbarkeit, Erweiterbarkeit, Wiederverwendbarkeit und Robustheit. Letztlich bieten sie eine Sicht auf das, was uns antreibt, wenn wir versuchen, gute Software zu bauen. Die vielen Maßnahmen und Instrumente des Softwareentwurfs wie z. B. Abstrahierung, Kapselung, Entkopplung usw. entspringen also den oben genannten Zielen. Bevor wir uns also den Maßnahmen und Instrumenten widmen, lohnt es sich, einen Blick auf diese grundsätzlichen Ziele zu werfen.

3.3.1 Wartbarkeit

Es gibt sicherlich Projekte, bei denen man weiß, dass ihre Laufzeit sehr begrenzt ist. Eine kleine unterhaltende Anwendung, die z. B. auf einer Messe einmalig gezeigt und danach nicht benutzt wird, hat also eine sehr begrenzte Laufzeit. Eine solche Anwendung muss nicht unbedingt besonders wartungsfreundlich sein, denn es handelt sich dabei mehr oder weniger um Wegwerfsoftware.

Viele andere Anwendungen hingegen haben eine sehr viel längere Laufzeit, mehrere Monate oder sogar viele Jahre. Solche Anwendungen werden unter Umständen sehr intensiv genutzt, vielleicht auch von einer großen Nutzermenge. Bei solchen Anwendungen bleibt es nicht aus, dass entweder immer wieder noch Fehler in der Anwendung zutage treten oder dass die Nutzer Änderungs- oder Erweiterungswünsche haben. Als Entwickler bedeutet dies, man wird solch ein Projekt nicht so schnell los und muss sich immer wieder damit beschäftigen, Teile der Anwendung ändern, Fehler ausmerzen und vieles mehr.

Dazu kommt, dass bei einer längeren Laufzeit einer solchen Anwendung die Chance groß ist, dass die Entwickler, die ursprünglich die Anwendung entwickelt haben, inzwischen nicht mehr im Unternehmen oder in der Agentur arbeiten. Es müssen also immer wieder neue Entwickler eingearbeitet werden. Eventuell ist die Anwendung sogar so konzipiert, dass Dritte die Möglichkeit haben, eigene Erweiterungen oder Veränderungen zu entwickeln, was die Sache noch zusätzlich verkompliziert. Gerade bei Projekten, wo nicht direkt eine Anwendung, sondern vielmehr eine Bibliothek entwickelt wurde, ist dies der Fall.

Das Warten und Betreuen bestehender Anwendungen oder Codeteile sind für die meisten Entwickler einfach nur lästig. Niemand möchte sich wirklich mehr mit den alten Kamellen beschäftigen, oft findet man im Nachhinein bestimmte Teile im Code nicht gut gelöst, oder man würde die Struktur mittlerweile ganz anders umsetzen. Sehr oft ist es auch einfach der Fakt, dass die Struktur mit der Zeit gewachsen und inzwischen immer schwieriger zu durchblicken ist.

All diese Probleme verschlechtern also die Wartbarkeit einer Anwendung oder eines Codeteils. Nun ist die Wartbarkeit aber in den seltensten Fällen eine konkrete Anforderung seitens des Kunden. Es ist vielmehr eine schwierig zu greifende Anforderung, die sich in vielen Details der Struktur und auch der konkreten Implementierung einer Anwendung widerspiegelt. Ich bezeichne es deswegen als eine sekundäre Anforderung.

Abbildung 3.7: Alter Anwendungscode kann manchmal schwer zu durchdringen sein.

Was macht nun aber eine gut wartbare Anwendung aus, wodurch wird eine Anwendung oder ein Codeteil gut wartbar? Man könnte ja ganz platt sagen, eine Anwendung, die fehlerfrei ist, ist auch automatisch gut wartbar, denn solange keine Änderungswünsche reinkommen, muss sie gar nicht gewartet werden. Das lassen wir aber nicht gelten, denn man kann nicht einfach eine Eigenschaft herstellen, in dem man sich Störfaktoren wegdenkt. Anwendungen sind nie völlig fehlerfrei. Vielmehr wäre Fehlerfreiheit ein wichtiges Merkmal, bevor die Anwendung überhaupt veröffentlicht wird. Danach müssen wir einfach davon ausgehen, dass ab und zu noch Fehler auftauchen werden.

Fragen wir uns erst einmal, was für Tätigkeiten wir bei der Wartung und Betreuung von Anwendungen eigentlich ausführen, um das Feld der Wartbarkeit ein wenig einzukreisen.

Lesbarkeit

Zuallererst müssen die für die Betreuung zuständigen Entwickler einmal verstehen, wie die Anwendung eigentlich aufgebaut ist und wie sie funktioniert. Hier ist natürlich klar, dass sie nicht zwingend sofort jedes Detail kennen müssen, aber es muss für sie möglich sein, mit so wenig Aufwand wie möglich die Funktionsweise eines Details der Anwendung kennenzulernen. Es geht hier also um Begriffe wie Übersichtlichkeit, Verständlichkeit, Lesbarkeit. Um optimale Wartbarkeit zu erzielen, müssen wir vom schlimmsten Fall ausgehen, also von einem Entwickler, der zuvor noch nichts mit der Anwendung zu tun hatte und sich dem Code zum ersten Mal nähert, bewaffnet nur mit einer groben Vorstellung von der Anwendung, die er vielleicht durch das Ausprobieren, sprich Ausführen der Anwendung erhalten hat, sowie vielleicht einer Handvoll Fachkonzepte und anderer Dokumente, von denen wir hoffen, dass sie wenigstens halbwegs korrekt und aktuell sind.

Was können die Anwendung, der Code und die Struktur nun tun, um übersichtlich, verständlich und lesbar zu sein? Letztlich zahlen die meisten Praktiken solider Softwareentwicklung nebenbei auch auf eine gute Lesbarkeit ein. Nehmen wir z. B. den Vorsatz, dass

eine Methode möglichst immer nur eine Aufgabe erfüllen soll und nicht mehrere Dinge implizit hintereinander. Dieser Vorsatz hat vordergründig erst einmal den Sinn, dass sich so eine bessere Entkopplung von Verantwortlichkeiten ergibt und außerdem eine sequenzielle Abhängigkeit innerhalb so einer Methode vermieden wird. Aber der angenehme Nebeneffekt ist auch, dass eine Methode, die einen sprechenden Namen hat und auch wirklich nur eine klar umrissene Aufgabe erfüllt, meist deutlich einfacher zu verstehen ist als eine Methode, die in sich nacheinander oder auch vermischt viele Dinge auf einmal macht. Ein guter Hinweis, dass man als Entwickler gerade dabei ist, solch eine Gemischtwarenladen-Methode zu schreiben, ist übrigens, wenn einem dafür partout kein vernünftiger Name einfallen will, der kürzer als 16 Zeichen ist.

Im Detail werden wir das noch in den folgenden Kapiteln besprechen, hier zunächst mal ganz grundsätzliche Punkte zum Thema gute Lesbarkeit:

Struktur, die das Konzept aufgreift: Wenn wir davon ausgehen, dass ein Entwickler von der Anwendung, die er betreuen soll, nur ihre äußerliche Funktionsweise kennt, weil er die Anwendung in Aktion gesehen hat und weil ihm ein fachliches Konzept zur Verfügung steht, dann hat er also auch nur eine fachliche Vorstellung von der Anwendung. Demzufolge ist es äußerst hilfreich, wenn sich die fachlichen Konzepte in der Struktur der Anwendung wiederfinden, denn so kann der Entwickler sofort eine gedankliche Verbindung zwischen den Dingen herstellen, die er aus dem fachlichen Konzept her kennt, und dem Code. Domain Driven Design ist hier einer der Schlüsselbegriffe. Die Anwendung wird also nicht primär nach technischen Gesichtspunkten strukturiert, sondern nach konzeptionellen, nämlich denen aus der Problemdomäne. Wenn es also in der Problem- oder Aufgabendomäne um Film, Videos und Ausleihverträge geht – wie in unserem Beispielprojekt FilmRegal –, dann würde es der Lesbarkeit dienen, wenn sich im Code der infrage kommenden Anwendungen diese fachlichen Objekte auch wiederfinden.

Sprechende Struktur, sprechende Namen: Eine einfach lautende Regel, aber oft nicht beherzigt. Paketnamen, Klassen, Interfaces, Methoden und Attribute sollten Namen haben, mit denen man auch etwas anfangen kann. Akronyme, Abkürzungen und dergleichen mögen den initialen Schreibaufwand verringern, aber Steve McConnell bringt es treffend auf den Punkt, wenn er sagt:

»*Programmcode wird viel öfter gelesen als geschrieben [...]. Sie sparen am falschen Ende, wenn Sie das Schreiben von Code erleichtern, aber das Lesen dadurch schwieriger wird.*« (McConnell 2007, S. 146)

Unter sprechenden Namen und einer sprechenden Struktur versteht man aber auch, dass diese Bezeichner einen wirklichen Aufschluss darüber geben, was die Verantwortlichkeit einer Klasse oder eines Interface ist oder was die Aufgabe einer konkreten Methode. Eine oft anzutreffende Eigenheit bei Flash-Anwendungen ist es zum Beispiel, dass Eventhandler nach dem Event bezeichnet werden, durch das sie aufgerufen werden, anstatt dass sie danach benannt werden, was sie tun. Ein Eventhandler mit dem Namen »onClick« verrät leider rein gar nichts darüber, was er eigentlich macht, sondern lediglich, wann er aufgerufen wird. Ein Entwickler muss nun also den gesamten Code durchforsten, um die Aufgabe der Methode herauszufinden. Zudem enthält die Bezeichnung eine stärkere Kopplung zur Klasse, die das Event wirft, denn wenn das click-Event später mal gegen ein press-Event

ausgetauscht werden soll, müsste eigentlich der Name des Eventhandlers geändert werden, obwohl sich eventuell in der Methode gar nichts ändert. Und ändert man den Namen nicht, wird die Verwirrung nur noch größer. Eine Lösung kann sein, dass man den Eventhandler neutraler benennt, z. B. »onButtonAction«. Zudem kann man im Eventhandler einfach eine weitere Methode aufrufen, die die eigentliche Aufgabe übernimmt. Ein Entwickler kann nun den Zusammenhang schnell erkennen zwischen einem aufgetretenen Event und der daraus erfolgenden Aktion.

Oft sieht man auch Konventionen, die einem Interface ein »I« voranstellen und die eine Klasse, die das Interface implementiert, dann ohne das I schreiben oder eventuell ein »Impl« hintanstellen. In solchen Fällen muss geprüft werden, ob nicht fachlich eine bessere Trennung von Interface und konkreter Klasse angebracht wäre, die eine bessere Benennung ermöglichen würde. Nehmen wir folgendes Beispiel:

Um Videos in Flash abzuspielen, können wir entweder direkt eine FLV Datei streamen, oder wir können ein Video in eine SWF-Datei einbetten und diese progressiv streamen oder ganz vorladen. Für beide Fälle könnten wir nun jeweils eine Klasse schreiben, die das eine oder das andere kontrolliert. Ein Videoplayer, der nun ein Video abspielen will, könnte nun entweder die eine oder die andere Klasse nutzen. Um die Benutzung zu vereinheitlichen, könnte der Videoplayer ein Interface definieren, in dem er angibt, wie er gerne ein Video benutzen möchte. Und die beiden oben genannten Klassen könnten das Interface implementieren, um dem Videoplayer dann ihre Dienste anbieten zu können. Nehmen wir an, wir bauen nun erst einmal das Interface. Wie sollen wir es nennen: »IVideo«? Und die FLVStream-Variante dann »VideoImpl«? Geht nicht, wie sollten wir sonst die SWF-Variante nennen.

Es besteht in diesem Fall kein Grund, das Interface mit einem I zu versehen. Der Videoplayer will Videos spielen, also nennt er sein Interface »Video«. Und die FLV-Variante benötigt kein Impl, sondern sie kann sich zum Beispiel »StreamVideo« nennen und die SWF-Variante dann »SWFVideo«. Nun mögen manche einwenden, dass durch diese Namen nicht klar wird, welches Interface diese Klassen implementieren. Wenn wir aber davon ausgehen, dass es normalerweise immer mehrere Klassen gibt, die ein Interface implementieren (sonst ist eventuell das Interface überflüssig), dann müssten diese Klassen folglich alle gleich heißen, was offensichtlich keine gute Strategie sein kann, denn nun wird einem nicht mehr klar, worin sie sich denn unterscheiden.

Dokumentation: Im Zusammenhang mit der Lesbarkeit von Code ist eine solide Dokumentation unerlässlich. Um die Wartbarkeit von Code zu erhöhen, ist es empfehlenswert, komplexere Bereiche im Code gesondert zu dokumentieren, damit andere Entwickler im Falle von notwendigen Änderungen oder bei der Fehlersuche schneller die Funktionsweise verstehen.

Darüber hinaus ist das Dokumentieren von Code mit JavaDoc-kompatiblen Kommentaren unerlässlich. Gerade zusammen mit heutigen IDEs wie Flex Builder und anderen werden diese während des Tippens dem Entwickler schon präsentiert und erleichtern die Entwicklung ungemein.

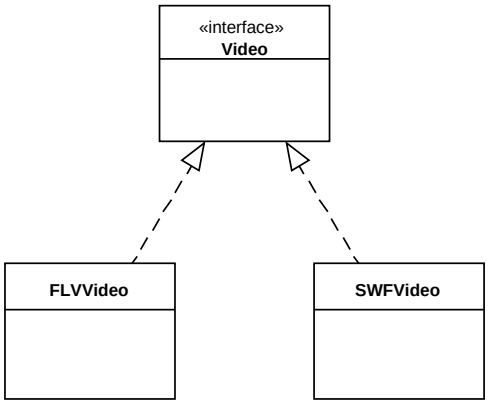


Abbildung 3.8: Benennung bei Interfaces. I ist nicht unbedingt notwendig.

Man kann natürlich auch zu viel dokumentieren. Eine Methode mit dem Namen getXPosition():int verrät im Prinzip alles Wissenswerte schon über ihre Deklaration, hier muss nicht mehr viel dokumentiert werden. In diesem Zusammenhang sei erwähnt, dass auch Typisierung enorm zur Lesbarkeit von Code beiträgt. Im obigen Beispiel verrät der Rückgabewert einiges darüber, was man von der Methode erwarten kann, z. B. dass nur ganze Pixelwerte ausgegeben werden. Würde man den Typ weglassen, würde einiges an Information verloren gehen.

Veränderbarkeit

Neben der Lesbarkeit spielt für die Wartbarkeit auch noch die Veränderbarkeit eine Rolle. Während des Lebenszyklus einer Anwendung wird an ihr immer wieder herumgeschraubt, auch wenn man das eigentlich ja vermeiden will. Der Grundsatz eines Softwareentwicklers heißt ja normalerweise: »Never touch a running system«, oder auch: »Repariere nichts, was nicht kaputt ist«. Aber über die Zeit verändern sich Anforderungen im Detail, die Arbeit mit der Anwendung enthüllt eventuell konzeptionelle Fehler oder auch technische Fehler, und die Folge ist, die Anwendung muss verbessert oder verändert werden.

Die Veränderbarkeit ist nun eine Eigenschaft, die sich in vielen Bereichen des Softwareentwurfs, aber auch der konkreten Implementierung niederschlägt. Im Entwurf ist die Modularität ein starkes Merkmal, das sich auf die Veränderbarkeit auswirkt. Je modularer, sprich stärker voneinander autark einzelne Module und Komponenten in der Anwendung gestaltet sind, desto einfacher kann man eine Änderung innerhalb eines Moduls machen, ohne dass sie sich zwangsläufig auf die anderen Module auswirkt. Und das ist ein hohes Gut bei der Veränderbarkeit: Änderungen machen zu können, ohne dass die ganze Anwendung gleich wie ein Kartenhaus in sich zusammenfällt.

Nehmen wir als Beispiel mal einen Parser, der eine Textdatei einlesen und seine Struktur in eine Objektstruktur parsen soll. Dieser Parser, der seinerseits Teil einer Anwendung ist, soll vielleicht optimiert werden, weil man gemerkt hat, dass er mit Textdateien ab einer gewissen Größe nicht mehr schnell genug vorankommt. Wenn nun dieser Parser eine klar defi-

nierte Schnittstelle hat, die seine innere Funktionsweise nach außen nicht preisgibt, dann stehen die Chancen gut, dass man seinen inneren Parsing-Algorithmus verändern kann, ohne etwas an der Schnittstelle nach außen ändern zu müssen. Das wiederum würde dazu führen, dass alle anderen Teile der Anwendung, die diesen Parser nutzen, nicht angefasst werden müssen, was ja das Ziel sein muss, um den Aufwand so gering wie möglich zu halten.

Was aber könnte man nun falsch machen, was würde zu einer schlechten Veränderbarkeit führen? Nun, der Parser könnte z. B. fälschlicherweise bestimmte Methoden oder Attribute, die Auskunft über seine innere Funktionsweise geben, öffentlich machen. Viele Parser, die Baumstrukturen parsen, verwenden rekursive Funktionen, um den Baum durchzugehen. Dabei übergibt man an die Funktion den aktuellen Knoten im Baum, und als Rückgabewert gibt die Funktion den geparsten Objektknoten zurück. Wenn diese Funktion public ist, besteht die Gefahr, dass sie von anderen Teilen der Anwendung vielleicht direkt verwendet wird, um »mal eben schnell« einen kleinen Teilbaum zu parsen. Dies führt nun dazu, dass es nicht mehr möglich ist, diese rekursive Funktion zu ändern oder gar gegen einen anderen nichtrekursiven Ansatz auszutauschen, weil er ja nun schon verwendet wird. Weiter unten im Abschnitt Kapselung werde ich über das Verstecken innerer Funktionalitäten einer Klasse noch detaillierter sprechen.

Testbarkeit

Einen letzten Bereich möchte ich im Bereich der Wartbarkeit noch ansprechen, die Testbarkeit. Logisch, wenn man eine Anwendung wartet und an ihr arbeitet, Fehler ausbessert, Optimierungen durchführt, muss man hinterher testen, ob auch noch alles funktioniert. Nun mag man sich fragen, was sich hinter Testbarkeit verbirgt. Testbar ist doch jede Anwendung, oder nicht? Mit Testbarkeit ist hier die Feingranularität gemeint, mit der man in einer Anwendung die einzelnen Teile zum einen isoliert voneinander und zum anderen im Zusammenspiel miteinander testen kann. Bleiben wir beim obigen Beispiel des Textparsers. Wir haben ihn verändert und müssen nun testen, ob die Anwendung auch immer noch genauso funktioniert, wie es gewünscht ist. Auch hier spielt die Modularität und Kapselung wieder eine große Rolle. Je weniger Abhängigkeiten der Parser zu anderen Teilen der Anwendung hat, je einfacher und klarer seine Schnittstellen sind, umso besser kann ich ihn isoliert testen. Wenn mein Parser beispielsweise noch fünf andere Klassen benötigt, um seine Arbeit zu verrichten, bedeutet dies, dass ich diese fünf anderen Klassen indirekt auch mittesten muss. Nun könnte man argumentieren, wenn man an diesen Klassen nichts verändert hat, können dort ja auch keine Fehler auftauchen. Die Veränderung im Parser kann aber sehr wohl Fehler in den anderen Klassen enthüllen, die früher nur einfach nicht in Erscheinung getreten sind, als der Parser noch anders gearbeitet hat.

Viele Abhängigkeiten zwischen Klassen erschweren das Testen bzw. das Finden von Fehlern, wenn welche auftreten. Klar voneinander getrennte Klassen oder Komponenten mit einfachen Schnittstellen erleichtern es hingegen enorm. Besonders deutlich wird dies, wenn man automatisierte Tests, z. B. Unit-Tests, schreibt. Bei Unit-Tests besteht ja immer die Möglichkeit, sogenannte Mock Objects zu schreiben für Klassen, die die zu testende Klasse eigentlich benötigt, die man aber nicht direkt mittesten will. Je mehr Mock Objects man

schreiben muss, um eine Klasse oder eine Komponente zu testen, desto eher ist das ein Anzeichen für eine schlechte Modularität. Und der Aufwand steigt natürlich immens. Denn diese Mock-Objekte simulieren ja andere Klassen. Wenn man aber nun diese anderen echten Klassen auch mal verändert, müssen die Mock-Objekte auch geändert werden. Das kann recht schnell in viel Arbeit ausarten.

3.3.2 Erweiterbarkeit

Flash gibt es zu der Zeit, in der dieses Buch entstanden ist, in Version 10. Die Flash API und auch die Sprache ActionScript selbst wurden stetig erweitert. Viele Anwendungen, die über einen längeren Zeitraum hinweg verwendet werden, werden um Funktionen erweitert. Erweiterungen sind dabei noch mal anders als Veränderungen, die wir im vorangegangenen Abschnitt behandelt haben. Bei Erweiterungen bestehen zwei gegensätzliche Herausforderungen. Einerseits muss die Anwendung die neue Erweiterung nutzen und ansprechen können, andererseits möchte man für eine Erweiterung möglichst wenig in der eigentlichen Anwendung verändern.

Nun könnte man sich eine Anwendung vorstellen, die als eine geschlossene Einheit aufgebaut ist, die in sich alle Funktionen erfüllt, die anfangs definiert wurden, und das war's. Eine solche monolithische Anwendung wäre dann im Prinzip gar nicht erweiterbar, ohne dass man sie nicht selbst stark abändert. Das Ziel ist ja aber, dass man eine Anwendung erweitern kann, ohne am bestehenden Code allzu viel ändern zu müssen. Das klingt nun nach einer nicht so einfachen Anforderung, denn man kann ja heute nicht wissen, was wohl in der Zukunft für Erweiterungen kommen mögen. Und eine Anwendung für alle Eventualitäten zu rüsten, würde einen immensen Aufwand bedeuten und den Code auch enorm aufblähen, was sich wiederum schlecht auf die Wartbarkeit auswirkt.

In der Tat ist die Erweiterbarkeit auch ein Merkmal, das nicht zwingend bei allen Anwendungen gleich wichtig ist. Anwendungen, die einen kurzen Lebenszyklus haben, müssen höchstwahrscheinlich nicht so stark erweiterbar sein wie Anwendungen, von denen man sich eine lange Lebensdauer erhofft. Wie kann man aber nun bei solchen länger laufenden Anwendungen für eine gute Erweiterbarkeit sorgen?

Erweiterbarkeit muss geplant werden. Man kann eine Anwendung nicht vorsorglich in alle Richtungen erweiterbar machen. In den meisten Fällen muss man das auch nicht. Nehmen wir als kleines Beispiel wieder den Videoplayer für unsere Firma FilmRegal. Wie können wir Erweiterbarkeit für einen Videoplayer planen? Was wollen wir erweiterbar halten bei einem Videoplayer? Zunächst fällt einem da das Videodatenformat ein. Seit einer bestimmten Version von Flash 9 können wir nicht mehr nur FLV-Dateien laden, sondern auch andere Dateiformate, solange sie einen kompatiblen Codec verwenden. Es ist denkbar, dass in der Zukunft weitere Formate dazukommen. Wenn das passiert, wäre es schön, wenn wir dazu nicht jedes Mal unseren Videoplayer komplett umbauen müssen. Das Beziehen der Videodaten sollte innerhalb des Players also vom Rest der Anwendung gekapselt werden, damit wir später weitere Formate hinzufügen können. Wir definieren also das Videodatenformat als einen Punkt, der erweiterbar sein soll.

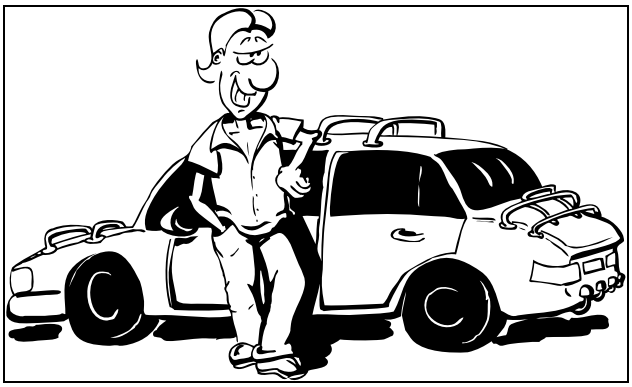


Abbildung 3.9: Enorme Erweiterbarkeit zum reinen Selbstzweck bringt meist wenig Nutzen.

Wie viele Stellen man in einer Anwendung für konkrete Erweiterbarkeit identifiziert, ist eine Abwägungsfrage zwischen Aufwand, Wahrscheinlichkeit des zukünftigen Wunsches der Erweiterung an dieser Stelle und negativen Einflüssen auf Wartbarkeit, Leistungsfähigkeit und anderen Eigenschaften der Anwendung. Es ist also letztlich in einem hohen Maß eine wirtschaftliche und strategische Überlegung. Strategisch auch, weil Erfahrungen mit ähnlichen Projekten aus der Vergangenheit Hinweise darauf geben können, wo sich in der Zukunft Anforderungen für Erweiterungen ergeben können. Bei einer Gruppe von Anwendungen, die in der Vergangenheit z. B. oft Änderungen und Erweiterungen mit einem bestimmten Backend-System erfahren hat, kann man davon ausgehen, dass dies ein Punkt für Erweiterungen auch in einem neuen Projekt sein kann, wenn es mit den anderen verwandt ist. In dem Fall sollte man auch in der neuen Anwendung dafür sorgen, dass entsprechend der Entwurf der Anwendung eine Erweiterbarkeit an der Schnittstelle zu besagtem Backend ermöglicht.

Um konkret geplante Erweiterbarkeit in einem hohen Maße zu erreichen, stehen uns Entwicklern einige nützliche Entwurfsmuster zur Verfügung, die speziell die Entkopplung von nutzenden zu genutzten Klassen modellieren, wie z. B. Decorator, Facade, Chain of Responsibility und andere, dazu in Kapitel 4.4 mehr.

Es gibt aber auch noch grundsätzliche Dinge, die die Erweiterbarkeit einer Anwendung begünstigen oder auch behindern. Modularität und Kapselung begünstigen die Erweiterbarkeit. Wenn in unserem Videoplayer die Anwendung keine direkte Kenntnis vom Videodatenformat hat, dann ist es überhaupt erst möglich, unterschiedliche Formate als Erweiterungen zu unterstützen. Wenn die Grundlogik des Videoplayers sauber getrennt ist vom User-Interface, dann kann später Letzteres um zusätzliche Bedienfeatures erweitert werden usw.

Je stärker wiederum einzelne Teile einer Anwendung konkrete Kenntnis von anderen Teilen haben, je verzweigter z. B. die gegenseitigen Methodenaufrufe sind – Klasse A verwendet zwei Methoden von Klasse B, die wieder mehrere Methoden von Klasse C, die wiederum einige von Klasse B und A usw. –, desto geringer ist zwangsläufig die Erweiterbarkeit einer solchen Anwendung.

Anwendungen also, die grundsätzlich stark modular und gekapselt aufgebaut sind, sind auch in den Teilen auf Erweiterungen gut vorbereitet, in denen nicht mit Erweiterungen gerechnet wurde. Man muss hier aber wie gesagt aufpassen, dass man einen gesunden Mittelweg zwischen Modularität und Sicherstellung von Leistungsfähigkeit und beherrschbarer Komplexität der Anwendung weiterhin gewährleistet, denn zu kleinteilige Modularität kann negative Auswirkungen auf andere Eigenschaften einer Anwendung haben, wie zum Beispiel Verständlichkeit und Robustheit.

3.3.3 Wiederverwendbarkeit

Dieses Prinzip spricht wunderbar unsere eigene Faulheit an. Softwareentwickler haben wie kaum ein anderer Beruf die Möglichkeit, Dinge, die sie einmal gebaut haben, wieder und wieder zu verwenden. Tischler, Maurer, Handwerker allgemein, sie alle müssen das, was sie einmal erfolgreich hergestellt haben, trotzdem immer wieder neu bauen. Sie können zwar Maschinen zu Hilfe nehmen, und ihre Erfahrung hilft ihnen auch, dass es beim nächsten Mal immer besser läuft, aber nichtsdestotrotz, ihre Erzeugnisse müssen für jeden Einsatzzweck immer wieder neu angefertigt werden.

Nicht so bei Software. Die simpelste Art der Wiederverwendung ist Copy&Paste. Jeder Entwickler hat das schon genutzt. Da hat man eine kleine Methode, vielleicht sogar nur eine einfache Schleife, und eh man die noch mal neu schreibt, kopiert man sie sich schnell. Was würde wohl ein Tischler geben, wenn er ein Stuhlbein nur einmal bauen und dann dreimal kopieren könnte!

In der Softwareentwicklung geht es aber noch viel besser. Jeder Entwickler, der schon ein wenig Erfahrung beim Programmieren gesammelt hat, weiß, dass das bloße Kopieren auf Dauer keine gute Lösung ist. Denn wenn man in dem kopierten Code hinterher noch einen Fehler findet, muss man ihn überall noch mal abändern. Stattdessen haben wir was viel Besseres. Anstatt etwas zu kopieren, können wir es in der Softwareentwicklung instanziieren und referenzieren. Wir würden unser virtuelles Stuhlbein also nur einmal bauen und nicht dreimal kopieren, sondern einfach vier Instanzen erstellen.

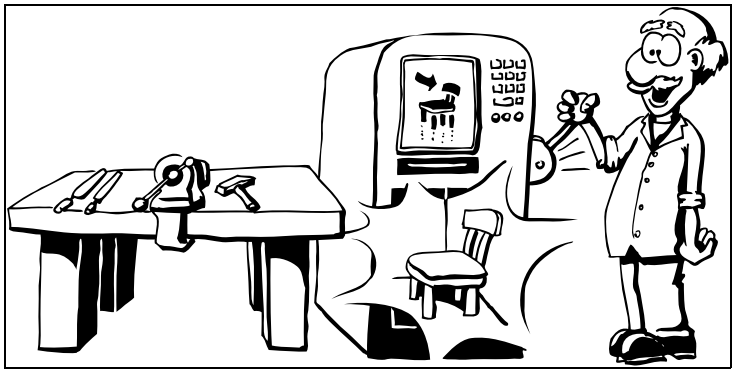


Abbildung 3.10: Einfache Instanziierung nach einem Bauplan geht nur in der Softwareentwicklung.

Wiederverwendbarkeit wird da kompliziert, wo sie plötzlich dynamisch und flexibel werden soll. Ich habe also Code schon mal geschrieben und könnte ihn eigentlich wiederverwenden, aber da ist ein kleines Detail, das brauche ich diesmal anders. Nehmen wir als Beispiel wieder mal unser Textparser-Beispiel von weiter oben. Nehmen wir an, als wir ihn geschrieben hatten, benötigten wir eine Klasse, die eine Textdatei ausliest und in eine Objektstruktur parst. Der Parser lädt also eine Datei über eine URL, die man ihm übergibt, parst dann seinen Inhalt und gibt die Objektstruktur zurück.

Nun habe ich eine andere Anwendung, die auch so eine Textstruktur parsen muss, die die Textdatei aus welchen Gründen auch immer schon fertig geladen vorrätig hat. Ich kann den vorhandenen Textparser nun leider so ohne Weiteres nicht verwenden, denn er funktioniert nun mal so, dass er selber die Textdatei laden will.

Dieses kleine Beispiel zeigt uns zwei Dinge. Will man wiederverwendbare Software schreiben, muss man einzelne Teile einer Anwendung losgelöst von ihrem ersten konkreten Einsatzort betrachten. Ein solcher Parser darf also nicht unter Berücksichtigung der konkreten Anwendung, für die man ihn baut, entworfen werden, sondern nur unter Berücksichtigung seiner ureigenen Funktion, nämlich das Parsen von Text. Es soll keine Abhängigkeiten zur konkreten Anwendung geben.

Daraus ergibt sich auch das Zweite: Will man wiederverwendbare Software bauen, müssen die wiederverwendbaren Teile in sich nur ihre ureigene Aufgabe erfüllen und keine andere. Ein Modul, das mehrere Verantwortlichkeiten in sich vereint, ist schwerer wiederverwendbar, denn die Chance, dass man beide Verantwortlichkeiten gemeinsam noch mal woanders benötigt, ist geringer als für die jeweiligen Verantwortlichkeiten allein. Die Chance also, dass ich einen Textparser noch mal woanders benötige, der auch gleichzeitig den Text lädt, ist geringer als bei einem Parser, der einfach nur Text parst.

Wir sehen also, die Krux liegt in der Granularität dessen, was ich als wiederverwendbar anbiete. Je stärker ein Codeteil sich auf seine eigentliche Aufgabe konzentriert, umso größer die Chance, dass ich ihn wiederverwenden kann. Wir merken schon, da schwingt wieder Modularität mit. Und weil das so ist, muss auch hier wieder darauf hingewiesen werden, dass eine hohe Wiederverwendbarkeit durchaus auch ihren Preis hat. Wenn man für jeden Codeschnipsel, für jede Klasse auf unbedingte Wiederverwendbarkeit achtet, erhöht man unter Umständen die Komplexität der Anwendung und verschlechtert damit eventuell die Wartbarkeit. Auch hier ist wieder eine Abwägung der verschiedenen Anforderungen und Erwartungen notwendig, um den richtigen Mittelweg zu finden.

Modularität ist auch nicht das einzige Mittel, um für eine gute Wiederverwendbarkeit zu sorgen. Wie schon oben erwähnt, sorgt Flexibilität für eine hohe Wiederverwendbarkeit. Komponenten, die sich in ihrer Funktionsweise an unterschiedliche Einsatzzwecke anpassen können, sind natürlich stärker wiederverwendbar als solche, die nur genau für einen Einsatzszenario geeignet sind. Variabilität ist also gefragt. Und das erhöht natürlich noch mal die Schwierigkeit bei der Erstellung von wiederverwendbaren Modulen. Nun muss man also nicht nur darauf achten, dass das Modul nur eine Verantwortlichkeit hat, man muss auch noch vorausahnen, in welchen Varianten es eventuell eingesetzt werden könnte.

Auf unseren Textparser übertragen könnte das zum Beispiel heißen, dass er mit unterschiedlichen Trennzeichen in der Datei – also Semikolon, Komma, Leerzeichen usw. – umgehen kann. Um wiederverwendbare Codeteile zu schreiben, muss man also von der konkreten Anwendung abstrahieren und sich vorstellen, in welchen Szenarien der konkrete Codeteil noch einsetzbar sein soll.

Auch hier gibt es wieder Entwurfsmuster, die uns helfen, Flexibilität und Variabilität in unseren zur Wiederverwendung vorgesehenen Code zu bringen, darunter z. B. Adapter, Mediator, Bridge und andere.

3.3.4 Robustheit

Bei vielen Flash-Anwendungen ist es wichtig, dass sie möglichst stabil laufen. Gerade bei Webanwendungen, die eher Content präsentieren als kritische Prozesse unterstützen, ist es im Zweifel wichtiger, dass eine Anwendung nicht abstürzt, als dass sie in Ausnahmefällen mal einen kleinen Fehler in der Anzeige hat oder unvollständige Daten präsentiert. Das soll nicht heißen, dass man seine Anwendung schon unter diesem Vorzeichen planen sollte, aber letztlich gilt es abzuwägen, ob eine Anwendung, wenn sie einen Fehler bemerkt, eher versucht, den Fehler abzuhandeln und weiterzumachen, oder ob sie sich beendet, weil der Fehler nicht vorkommen darf, wie es z. B. in sicherheitskritischen Anwendungen ja durchaus der Fall sein kann.

Eine Anwendung gilt als robust, wenn sie sich auch in Fehlersituationen wieder fängt und weiter arbeiten kann. Zur Robustheit gehört also zu einem großen Teil, mit Fehlern tolerant umzugehen. Und vor allem, Fehler überhaupt zu erkennen und abzufangen. Eine Anwendung gilt wiederum als »korrekt«, wenn sie in einem Fehlerfall nicht mit fehlerhaften Daten weiterarbeitet, sondern sich eher beendet und so gar nicht erst die Option eröffnet, mit falschen Daten oder in einem nicht validen Zustand weiterzuarbeiten.

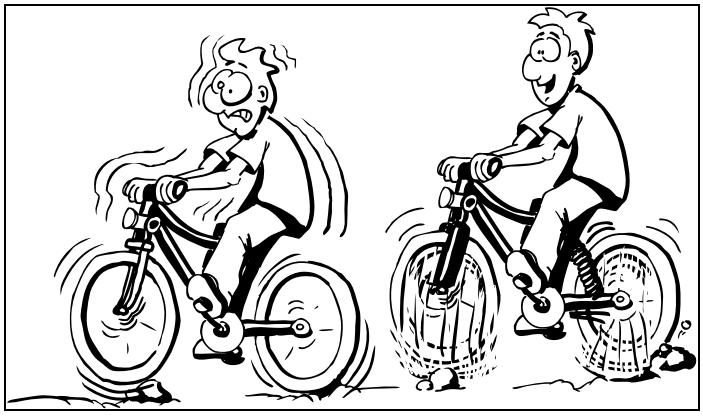


Abbildung 3.11: Eine robuste Anwendung kann Fehler abfangen.

Eine große Gefahr bei sehr modularen Anwendungen, die vielleicht in einem großen Team erstellt wurden, bei dem jedes Teammitglied jeweils ein anderes Modul gebaut hat, ist folgende: Sobald man alle Module zur gewünschten Anwendung zusammenführt, kann alles in sich zusammenstürzen. Hohe Modularität kann unter Umständen eine schwache Robustheit einer Anwendung zur Folge haben, wenn nicht die einzelnen Module schon sehr robust sind. Modulare Anwendungen haben die Eigenart, dass sich jedes Modul natürlich nur auf seine Aufgabe konzentriert und zu Recht erwartet, dass die anderen Module ebenso ihre Aufgabe erfüllen. Was aber, wenn ein Modul mal seine Aufgabe nicht erfüllen kann, weil es entweder fehlerhaft ist oder weil bestimmte Bedingungen, unter denen es normalerweise arbeitet, nicht gegeben sind?

Was z.B., wenn unser Textparser von weiter oben nicht validen Text zum Parsen bekommt? Zur Robustheit gehört eine gründliche Fehlerbehandlung, und zwar in zweierlei Hinsicht. Module müssen frühzeitig nach außen melden, wenn sie falschen Input bekommen oder in sonstiger Hinsicht in einer nicht unterstützten Art und Weise verwendet werden. Auf der anderen Seite müssen Module, die andere Module benutzen, damit rechnen, dass diese anderen Module fehlerhaft sein könnten oder aus sonstigen Gründen nicht das tun, was sie sollen.

Eine Klasse also, die z. B. den Textparser verwendet, muss in dem Fall, dass der Parser nicht wie gewünscht eine Objektstruktur zurückliefert, eine Strategie parat haben, wie sie selbst weiterarbeitet. Das kann heißen, dass sie eventuell die benötigten Daten von einem anderen Modul beziehen kann, oder aber im schlechtesten Fall, dass sie ihrerseits nach außen melden muss, dass ein Fehler aufgetreten ist.

Seit Flash 9 gibt es für das Melden von Fehlern zwei offizielle Wege. Zum einen können für synchrone Fälle die klassischen Errors mittels »throw« geworfen werden. Zum anderen können in asynchronen Situationen Error-Events dispatched werden. Werden solche Error-Events nicht durch einen Eventhandler abgefangen, hat dies den gleichen Effekt wie das Werfen eines normalen Errors. Um eine robuste Anwendung zu schreiben, sollte also jedes Modul entsprechende Fehlerbehandlungen implementieren.

3.4 Abstrahierung

Ich habe schon in der Einleitung von Kapitel 3 von der außerordentlichen Fähigkeit von uns Menschen zur Abstrahierung gesprochen. Der große Vorteil in dieser Fähigkeit liegt darin, dass wir uns von den konkreten Ereignissen und Zuständen unserer Umwelt Modelle im Kopf zurechtlegen. Wir merken uns das Konzept eines Autos nicht dadurch, dass wir ein ganz konkretes Auto abspeichern, sondern dass wir ein abstraktes Modell eines Autos konstruieren, das künftig auf alle Autos passt, die wir sehen. Das tun wir ganz intuitiv mit allen Dingen, die wir sehen und denen wir begegnen.

Diese Abstrahierung hat viele Vorteile. Zum einen ist es z. B. viel einfacher, mit anderen Menschen über abstrakte Dinge zu sprechen als über konkrete. Als unser Fahrlehrer die Funktionsweise eines Autos erklärt hat, saßen wir zwar in einem konkreten Auto, unser Fahrlehrer hat aber nicht jedes Mal konkret über dieses Auto gesprochen, sondern ganz all-

gemein von Lenkrad, Bremspedal, Gangschaltung usw. Und wenn wir dann mit Freunden übers Autofahren gesprochen haben, haben wir immer »Auto« gesagt und nicht »Gefährt mit vier Rädern, Verbrennungsmotor, Fahrgastzelle und Schalensitzen«. Abstrahierung heißt hier also auch, dass wir komplexe Dinge unter einem abstrakten Begriff fassen können.

Neben der vereinfachten Kommunikation bedeutet Abstrahieren aber auch, dass wir das Wissen über grundsätzliche Eigenschaften und Funktionsweisen von einer konkreten Sache lösen und uns mittels eines abstrakten Objekts vorstellen können, nur um sie in einer anderen Situation wieder auf eine konkrete Sache der gleichen Art anzuwenden.

Obwohl wir nämlich das Fahren in einem ganz konkreten Auto erlernt haben, sind die meisten von uns in der Lage, in so gut wie jedes Auto einzusteigen, sich kurz umzusehen und loszufahren. Wir sind also nicht nur in der Lage, konkrete Eigenschaften und Funktionsweisen von einem konkreten Ding zu lösen und uns abstrakt zu merken, wir können diese Eigenschaften und Funktionsweisen auch wieder auf ein ganz konkretes anderes Objekt übertragen, wenn es grundsätzlich mit unserem abstrakten Modell in Einklang gebracht werden kann. Fahren lernen im Fahrschulwagen, durch die Gegend fahren dann im ersten eigenen Wagen.

Um dieses Beispiel noch stärker zu belasten, schauen wir uns die Abstrahierung auch noch aus einem dritten Blickwinkel an. Und zwar sehen wir da die Autohersteller, die ihre Autos so konstruieren, dass sie unserer abstrakten Vorstellung von einem Auto so nah wie möglich kommen, damit wir überhaupt in der Lage sind, ihre Autos zu fahren. Der Autohersteller muss von der allgemein bekannten abstrakten Vorstellung von einem Auto ausgehen, wenn er seine konkreten Autos entwickelt. Er geht also von den abstrakt formulierten Eigenschaften und Funktionsweisen aus, die wir einem Auto zuordnen, und implementiert sie in konkreter Form in seine Autos. Wie sehr uns auch nur die kleinste Abweichung auffällt, merkt man, wenn man einem Elektroauto auf der Straße begegnet ist und irritiert war, dass es keine nennenswerten Geräusche von sich gibt, was so weit geht, dass Hersteller solcher Autos mittlerweile optional künstliche Motorgeräusche erzeugen, damit sich unsere abstrakte Vorstellung von einem Auto mit dem konkreten Fahrzeug wieder deckt.

Abstrahierung und Modellierung ist also nichts Neues für uns, im Gegenteil, es ist ein Instrument, das wir intuitiv von Geburt an praktizieren. Wo liegt hier also das Problem, gibt es überhaupt eins? Die Probleme fangen dort an, wo wir die abstrakten Eigenschaften und Funktionsweisen einer Sache nicht gleich erkennen oder wir sie zunächst falsch deuten.

Eines der berühmtesten Beispiele für die Schwierigkeit, das abstrakte Modell zu erkennen, ist sicherlich das Universum. Richtig erkannt haben wir es immer noch nicht, obwohl wir uns in unserer Geschichte schon diverse Modelle erdacht haben, um den Sternenhimmel zu beschreiben. Da war das Himmelszelt, an dem die Sterne angeheftet waren, da war das geozentrische System, bei dem die Erde im Mittelpunkt stand und sich alles drum herum bewegte. Da war Atlas, der die Erde stemmte, und natürlich die biblische Anschauung, in der die Erde und das Universum von Gott in sieben Tagen erbaut wurden. Heute ist die Urknalltheorie populär und die dunkle Materie. Wer weiß, welche neuen Thesen wir uns in der Zukunft noch ausdenken werden.

Das Bemerkenswerte hierbei ist, dass die Modelle, die wir Menschen uns bezüglich des Universums über die Jahrtausende ausgedacht haben, jeweils nie schlecht waren, denn sie haben in dem Kontext, in dem sie entwickelt wurden, immer ihren Zweck erfüllt. Die Modelle reichten aus, dass schon vor mehreren Tausend Jahren unsere Vorfahren in der Lage waren, Jahreszeiten zu bestimmen und Aussaat und Ernte festzulegen. Die Gravitationsmodelle, die sich Newton erdacht hat, können nicht alle Phänomene erklären, die wir heutzutage zu beobachten imstande sind, aber sie haben über mehrere Jahrhunderte ein ausreichend genaues Modell über die Verhaltensweisen der Planeten im Weltraum beschrieben, mithilfe derer viele Probleme und Fragen gelöst werden konnten.

Die Modelle wurden also nie aus reinem Selbstzweck entwickelt, sondern um ganz bestimmte Aufgaben zu meistern. Wir wissen heute, dass es nicht zwingend notwendig, ja vielleicht gar nicht möglich ist, das vollständig korrekte Modell des Universums zu beschreiben. Es reicht aus, wenn wir ein Modell haben, mit dem wir die Aufgaben, die wir heute erledigen wollen, lösen können. Und wenn sich neue Probleme ergeben, dann müssen wir halt gegebenenfalls unser Modell anpassen. Diese Erkenntnis ist auch in der Softwareentwicklung wichtig. Um in einem Problembereich eine gute Anwendung zu entwickeln, müssen wir nicht ein vollständiges und exaktes Modell des Problems erstellen. Es reicht, wenn wir ein Modell erstellen, das so genau ist, dass wir damit unser Problem gelöst bekommen. Genau das erreichen wir mit unserer Fähigkeit zur Abstrahierung, denn Abstrahieren heißt auch, die Details wegzulassen, die uns im Rahmen unserer Aufgabe nicht interessieren. Und das hilft uns letztlich wieder, die Komplexität der Aufgabe zu reduzieren.

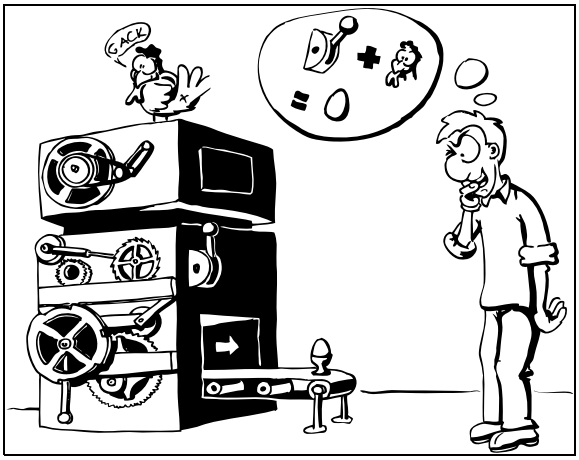


Abbildung 3.12: Hebel plus Henne ergibt ein Ei. Wie die Maschine funktioniert, muss uns nicht interessieren.

Auch in der Softwareentwicklung stehen wir immer wieder aufs Neue vor der Aufgabe, ein abstraktes Modell eines Problems zu erstellen. Selbst schon bei so vermeintlich kleinen Anwendungen wie unserem Videoplayer unserer Beispielfirma FilmRegal haben wir damit

zu tun. Schon der Name Videoplayer ist in diesem Fall doppelt abstrakt. Schon in der Realität bezeichnet der Begriff auf abstrakte Weise eine Gruppe von Geräten, die früher VHS-Kassetten, heute eher DVDs und Blueray-Disks abspielen. Zig Hersteller bauen die unterschiedlichsten Varianten dieser Player, und doch wissen wir einigermaßen genau, was so ein Gerät zum Videoplayer macht. Die doppelte Abstraktion kommt nun zustande, weil wir diesen eh schon abstrakten Begriff in die Softwarewelt hieven, wo er nur noch in seiner Funktionalität und seiner Grundidee weiterexistiert, aber nicht mehr in seiner Form, denn unser Software-Videoplayer ist nun virtuell.

Wenn wir uns heutige Computer ansehen, stellen wir fest, dass sich fast alle Computerprogramme abstrakter Begriffe aus der realen Welt bedienen, um ihren Nutzern ihre Funktionalität in einem vertrauten Umfeld darzustellen. Betriebssysteme arbeiten mit Fenstern, Knöpfen, Zeigern, Schiebern, mit Listen, Tabellen, Ordnern usw. Alles Begriffe, die aus der realen Welt abstrahiert wurden. Da sie in der virtuellen Welt ihr funktionales Konzept behalten haben (in einen virtuellen Ordner kann man z. B. wie in einen echten Ordner Unterlagen ablegen, mit Knöpfen aktiviert man eine bestimmte Aktion), finden sich die Nutzer mit ihnen schnell zurecht, weil die Nutzer das gleiche abstrakte Verständnis von diesen Dingen haben.

Im objektorientierten Entwurf versucht man nun konsequent, diese Abstraktion bis ins Detail durchzuhalten. Dabei wird die Abstrahierung meistens da immer schwieriger, wo man sich in Bereiche begibt, für die es noch keine gängigen Abstrahierungen gibt. Schauen wir uns das wieder am Beispiel unseres Videoplayers an. Die ersten Abstrahierungen sind noch einfach. Bei einem Videoplayer sprechen wir von einem Video, dem Nutzer, der Bedienleiste, den Nutzungsrechten, die der Nutzer benötigt, damit er das Video ansehen darf, usw. Diese Begriffe sind eingängig, denn jeder kann etwas damit anfangen. Je mehr wir aber in den technischen Bereich kommen, umso schwieriger wird die Abstrahierung. Beispielsweise wollen wir das Video nicht erst komplett herunterladen, sondern schon während des Ladens abspielen. Das ist ein recht spezieller technischer Vorgang, für den es in der realen Welt kaum übertragbare Vorgänge gibt, von denen man abstrahieren könnte. Uns fällt es hier natürlich deutlich schwerer, sinnvolle Abstrahierungen zu finden. Glücklicherweise haben sich andere Entwickler dazu schon Gedanken gemacht und haben das kontinuierliche Laden und Auswerten von Daten mit einem Strom verglichen, einem Datenstrom, der vom Server zum Client fließt und den wir direkt dort abgreifen können. Deswegen bietet uns die Flash API auch die NetStream-Klasse an. Sie baut auf dieser abstrakten Idee eines Datenstroms auf.

Nun könnte man sich fragen, warum es denn so wichtig ist, solcherlei Abstrahierungen zu finden. Warum verwendet man nicht einfach irgendwelche technischen Namen? Wichtig ist zunächst einmal, dass es bei der Abstrahierung nicht vordergründig um die Namen geht. Die Idee des Datenstroms ist nicht deswegen gut, weil Strom so ein tolles Wort wäre oder weil man sich bei einem Strom so schön fließendes Wasser vorstellen könnte. Die Idee ist gut, weil das Konzept des Stroms abstrakt genug ist, um jeglichen kontinuierlichen Transport von Daten von einem System X zu einem System Y zu bezeichnen. Das eignet sich nicht nur für Videostreams, die von einem Server kommen, sondern auch für File-Streams, bei denen eine Datei kontinuierlich von einer Festplatte gelesen wird, oder Audiodaten, die von einem Mikrofon erzeugt werden. Bei der Abstrahierung kommt es also nicht zwingend

auf einen guten Namen an – obwohl ein sprechender Name immer hilfreich ist, siehe Lesbarkeit –, sondern auf eine Idee, die abstrakt und tragfähig genug ist, um eine Sache auf einem allgemeinen Level zu beschreiben, sodass man davon wieder konkrete Implementierungen ableiten kann.

Trotzdem bleibt die Frage, ob dies immer notwendig ist. Muss jeder Teil einer Anwendung eine konkrete Implementierung eines abstrahierten Konzepts sein? Sicherlich nicht. Auch hier ist wieder ein Mittelweg gefordert. Geeignete Abstrahierungen zu finden, ist immer dann besonders nützlich, wenn man ein technisches Konzept fit für eine generelle Wiederverwendung machen will oder wenn man schon weiß, dass ein Programmteil auf einem bereits existierenden abstrakten Konzept basiert. Wenn man also einen Videostream implementiert (nehmen wir mal für einen Moment an, die Flash API würde einen solchen nicht schon konkret anbieten), dann ist es sinnvoll, auf dem schon existierenden abstrakten Konzept des Datenstroms aufzusetzen, weil damit gewährleistet ist, dass andere Entwickler, denen dieses Konzept vertraut ist, auch mit dem Videostream schnell klarkommen werden. In der Praxis wird man mit der Zeit feststellen, dass bereits für sehr viele technische Konzepte entsprechende Abstraktionen gefunden wurden. In der Regel wird man für seine konkrete Implementierungsidee ein bereits bestehendes abstraktes Konzept finden, mithilfe dessen man seine Idee standardisieren kann. Hier hilft z. B. oft ein Blick in bestehende API-Beschreibungen, z. B. die von Flash und Flex, aber durchaus auch die von anderen Plattformen, wie z. B. Java. Die Abstrahierung gilt übrigens nicht nur für konkrete Klassen, sondern auch für Klassenstrukturen. Entwurfsmuster sind letztlich auch Abstrahierungen von konkreten Klassenstrukturen, die bestimmte strukturelle Lösungen anbieten.

Bisher haben wir über Abstrahierung nur im theoretischen Rahmen gesprochen. Wie aber stellt sie sich konkret im Code dar? In welcher Form findet sich Abstraktion in Anwendungen konkret wieder? Die Antwort darauf fällt schwer, denn Abstraktion erscheint in vielen unterschiedlichen Formen im Code. Diejenige, die einem sofort einfällt, ist natürlich die Vererbung, auf die ich noch zu sprechen kommen werde. Aber auch das Interface-Konzept fußt letztlich auf der Idee der Abstraktion, genauso wie Pakete, eine konkrete Klasse und sogar eine Methode. Schauen wir uns das an.

Die Paketstruktur – oder auch die Struktur von Namespaces, wie ActionScript 3 sie heutzutage gestattet – bietet eine Möglichkeit der Abstraktion. Wenn ich innerhalb unseres Videoplayers ein Paket mit dem Namen user erstelle und dort alle Klassen und Interfaces, die sich um den Nutzer drehen, versammle, dann bietet allein dieses Vorgehen eine grundsätzliche Abstraktion, denn über diese Strukturierung mache ich allein über ein Paket nach außen kenntlich, dass hier das Konzept eines Nutzers implementiert ist, ohne dass man sich zu diesem Zeitpunkt schon mit Details darüber befassen müsste. Natürlich sind Pakete allein ein recht schwaches Instrument der Abstrahierung, denn sie sagen außer dem Namen nicht sehr viel mehr über das Wesen der Sache aus, die dort behandelt wird.

Als Nächstes haben wir die Vererbung, ein sehr starkes Instrument zur Abstraktion. Über sie wird deutlich, welchen technisch konzeptionellen Ursprung eine Klasse haben kann. So wird dadurch, dass in der Flash API ein Button letztlich indirekt von DisplayObject erbt, deutlich, dass ein Button nun mal ein visuelles Objekt ist, das die gleichen Grundeigenschaften besitzen sollte wie jedes anderes visuelle Objekt auch.

Interfaces unterstützen Abstrahierung auf eine andere Art und Weise. Anstatt wie bei der Vererbung auszudrücken, dass ein Button ein DisplayObject *ist*, würde man auf Interfaces übertragen sagen, dass sich ein Button wie ein DisplayObject *verhält*. Allein die Formulierung lässt erahnen, dass die Idee der Interfaces noch etwas lockerer ist als bei der Vererbung. Denn ein Objekt, das sich nur wie ein anderes verhält, gibt eben nicht preis, was es letztlich wirklich ist, sondern nur, was es sein kann. Das wird einem klarer, wenn man sich vor Augen hält, dass eine ActionScript-Klasse ja mehrere Interfaces implementieren, aber nur von einer anderen Klasse erben kann. Die Idee der Interfaces ist also etwas freier. Mit ihnen lassen sich deswegen besonders gut entkoppelte Modulstrukturen realisieren.

Als Nächstes haben wir die konkrete Klasse. Selbst sie stellt eine Abstrahierung dar. Denn eine Klasse präsentiert sich gegenüber anderen Klassen in Form ihrer öffentlichen Schnittstelle. Unser Textparser von weiter oben z. B. hat vielleicht einfach eine öffentliche Methode `parseText(text:String):ObjectStructure`. Intern wiederum verwendet er unter Umständen weitere Methoden, besitzt mehrere Attribute und verwaltet Hilfsobjekte. Abstrahierung meint in diesem Zusammenhang, dass wir, die den Textparser benutzen, nur das Konzept des Parsens von Text in der Form »Text rein, Objektstruktur raus« verstehen müssen und nicht die Details der Implementierung. Das konkrete Vorgehen wird also abstrahiert, und übrig bleibt die oben genannte Methode. Klassen sind also wichtige grundsätzliche Werkzeuge der Abstrahierung von Elementen einer Anwendung.

Auf dem kleinsten Level haben wir schließlich die Methoden. Auch eine einzelne Methode abstrahiert. Nehmen wir eine der ursprünglichsten Methoden von Flash überhaupt: `gotoAndPlay()`. Wir können uns sicher sein, dass der Aufruf dieser Methode der MovieClip-Klasse innerhalb des FlashPlayers einiges bewirkt. Aber die Methode abstrahiert von den vielen kleinen Einzelschritten das Grundkonzept des Springens in einen anderen Frame der Timeline (und des Weiterspielens) und erleichtert uns damit den Umgang mit dieser vermeintlich komplexen Funktionalität. Methoden sind deswegen die kleinstmögliche Form der Abstrahierung, die wir zur Verfügung haben.

3.5 Kapselung

Stellen Sie sich vor, in Ihrem Auto wären sämtliche veränderlichen Eigenschaften direkt vom Fahrersitz aus zugänglich, also nicht nur Lenkung, Gas und Bremse, sondern auch Verdichtungsverhältnis, Zündzeitpunkt, Steuerung des Motorlüfters und viele andere. Technikfreaks würden sich darüber vielleicht sogar freuen, weil es ihnen enorm viele Einstellungsmöglichkeiten ermöglicht. Aber grundsätzlich würde diese öffentliche Zugänglichkeit die Gefahr bergen, dass man aus Versehen Einstellungen vornimmt, die entweder die Leistung des Autos beeinträchtigen oder sogar das Fahrzeug zum Stillstand bringen. Auch würden diese vielen Einstellungsmöglichkeiten die meisten Fahrer überfordern, weil man vor lauter Schaltern und Reglern kaum noch die wichtigen von den unwichtigen Funktionen unterscheiden könnte.

Einen Flugschein haben nicht zuletzt deswegen relativ wenige Menschen, weil das Erlernen der vielen Funktionen in einem Flugzeug plus die umfangreichen Regeln im Flugverkehr

sehr aufwendig und deswegen auch teuer ist. Im Flugzeug kapseln die Instrumente im Cockpit zwar auch Funktionalitäten und Vorgänge, es werden aber immer noch deutlich mehr Steuerungsmöglichkeiten und Informationen an den Piloten weitergegeben als an einen Autofahrer. Das Autofahren hat auch deswegen seinen Siegeszug durch die Länder der Welt angetreten, weil es relativ einfach zu erlernen ist, denn die Funktionen und Eigenschaften eines Autos, die man zum Fahren zwingend kennen muss, sind überschaubar.

Wie wir schon im vorigen Kapitel gelernt haben, können wir die Komplexität einer Sache durch Abstrahierung verringern. Indem wir uns nur auf die für die Betrachtung einer Sache wichtigen Dinge konzentrieren und alles andere außer Acht lassen, verringern wir die Komplexität und haben eine Chance, den Sachverhalt zu verstehen.

Die Kapselung ist nun das praktische Instrument zur Durchsetzung der Abstrahierung. Im Auto findet die Kapselung über die Trennung von Innen- und Motorraum statt. In den Innenraum ragen nur die wichtigen Funktionen hinein, wie eben die Pedale und das Lenkrad. Zudem geben uns die Instrumente nur eine abstrakte Sicht auf das, was einen knappen Meter weiter vorne tatsächlich passiert. Mit der Kapselung im Softwareentwurf verstecken wir explizit bestimmte Codeteile vor anderen Codeteilen bzw. den Entwicklern dieser anderen Teile. Indem wir nur die Teile unseres Codes zugänglich machen, die ein anderer Entwickler auch verwenden soll, ersparen wir ihm, sich durch einen Wust an Code durcharbeiten zu müssen, bevor er versteht, welche Funktionen für ihn wirklich interessant sind.

Das klingt natürlich ein wenig nach Bevormundung, und in der Tat ist das auch ein bisschen so. Aber letztlich hilft es den einzelnen Entwicklern, sich nicht mit unzähligen Funktionen und Eigenschaften beschäftigen zu müssen, sondern sich auf die wenigen Funktionen konzentrieren zu können, die für das Benutzen wichtig sind. Kapselung ist somit eine praktische Methode zur Verringerung von Komplexität und auch zur Verringerung von Fehlerquellen, denn was ich nicht benutzen kann, kann ich auch nicht falsch benutzen. Sehen wir uns hierzu ein kleines Beispiel an.

```
package {

   import flash.geom.Point;

   public class Headline extends Sprite {

      private var xOffset:Number = 0;
      private var yOffset:Number = 0;

      public static var H1:Point = new Point( -4, -3);
      public static var H2:Point = new Point( -2, -2);
      public static var H3:Point = new Point( 0, -1);

      function Headline(size:Point) {

         setOffset(size.x, size.y);
         refresh();
      }
```

```
        public function setOffset(i_xOffset:Number, i_yOffset:Number):void {

           xOffset = i_xOffset;
           yOffset = i_yOffset;
        }

        private function refresh():void {

           // Aktualisiert das Layout der Headline ...
        }
     }
  }
```

Listing 3.1: Beispiel für schlechte Kapselung

Das Beispiel in Listing 3.1 zeigt eine einfache – nicht vollständige – Headline-Komponente. Wir gehen davon aus, dass diese Komponente intern ihr Textfeld je nach gewählter Schriftgröße ein wenig zurechtrücken muss, damit es visuell korrekt auf dem Nullpunkt der Komponente sitzt. Dies soll angedeutet werden durch die beiden Werte `xOffset` und `yOffset`. Betrachten wir diese Klasse nun aber nur aus dem Blickwinkel der Kapselung. Zunächst einmal sind die beiden Offset-Werte private, was schon mal positiv auffällt, denn so sind sie nicht direkt zugänglich und können nicht fälschlicherweise direkt verändert werden.

Die drei vermeintlichen Konstanten allerdings sind nicht mittels `const` wirklich als Konstanten definiert, sondern per `var` veränderlich. Das ist an sich keine gute Kapselung, denn nun könnte man den Inhalt der Konstanten ändern, was bei Konstanten eigentlich nicht gemacht werden soll. Viel schlimmer aber ist, dass sie vom Typ eines Points sind und konkrete Offset-Werte enthalten. Das hat zwei Nachteile: Zum einen würde das Verwenden von `const` für diese drei Attribute keinen großen Unterschied machen, denn die Werte im Point wären nach wie vor veränderbar. Zum anderen wird nach außen ein wichtiges Implementierungsdetail dieser Klasse preisgegeben, nämlich dass sich hinter den Headlinegrößen scheinbar Steuerungswerte verbergen.

Der Konstruktor verlangt als Parameter ein Point-Objekt. Nehmen wir mal an, die Dokumentation klärt uns Entwickler darüber auf, dass wir eine der drei Quasi-Konstanten übergeben sollen. Die Tatsache, dass wir schon wissen, was sich hinter den Konstanten verbirgt, versetzt uns nun in die Lage, einfach ein Point mit zwei beliebigen Werten an den Konstruktor zu übergeben und so die schöne Idee mit den festen drei Headlinegrößen zunichte zu machen. Aber das war noch nicht alles. Der Konstruktor ruft intern zunächst die `setOffset()`-Methode auf und danach die `refresh()`-Methode. Jemand war zwar so schlau, die `refresh()`-Methode private zu deklarieren, aber die `setOffset()`-Methode ist public. Und sie nimmt ganz konkret einen Wert für den X-Offset und für den Y-Offset entgegen. Wenn es vorher keinem klar war, dann spätestens jetzt, wie die Klasse bezüglich der unterschiedlichen Größen arbeitet.

Warum ist das aber eigentlich so schlimm? Was ist schon dabei, wenn man einige Details der Implementierung der Klasse kennt? Könnte es nicht vielleicht sogar nützlich sein? Der erste Schwachpunkt dieser Offenheit ist, dass sich für den Entwickler, der diese Klasse nutzt, klammheimlich die Komplexität der Klasse erhöht. Da die `setOffset()`-Methode nun

schon für ihn zugänglich ist, ist die Sache der Offsets eine Information, mit der er zwangsläufig konfrontiert wird, obwohl ihm eigentlich egal sein kann, wie die Headline-Komponente es nun genau anstellt, sich korrekt zu positionieren. Man muss hier bedenken, wir gehen von einem Szenario aus, bei dem mehrere Entwickler an einem Projekt arbeiten und unterschiedliche Teile einer Anwendung bauen. Hätten wir nur einen einzelnen Entwickler, der für sich allein ein Projekt stemmt, treten die Probleme nicht direkt in Erscheinung. Wir wollen also verhindern, dass sich ein Entwickler in einem Team mit zu viel Details von Anwendungsteilen beschäftigen muss, die von anderen Entwicklern im Team stammen. Je weniger die Headline-Komponente also unnötigerweise von sich preisgibt, desto übersichtlicher und einfacher stellt sie sich für einen Entwickler dar.

Darüber hinaus wird auch die Fehleranfälligkeit verringert. Nehmen wir mal an, ein Entwickler hat sich die öffentlichen Methoden der Headline-Klasse angesehen und denkt sich nun, er könnte ja `setOffset()` einfach mit eigenen Werten aufrufen, weil er eine ganz spezielle Headlinegröße benötigt. Nun hätte er das Problem, dass ja der Konstruktor nach Aufruf von `setOffset()` noch `refresh()` aufruft. Die Methode ist aber private, die kann von außen gar nicht aufgerufen werden. Der manuelle Aufruf von `setOffset()` wäre also zum Scheitern verurteilt. Bis der Entwickler bemerkt, dass er `setOffset()` eigentlich gar nicht selbst benutzen sollte, kann aber einige wertvolle Zeit vergehen.

Es gibt noch ein weiteres Problem. Da über die beschriebenen Merkmale nun die Implementierung der Headlinemechanik mittels Offsets bekannt ist und manch ein Entwickler sich das eventuell zunutze macht und im Konstruktor ein Array mit eigenen Werten mitgibt, ist es dem Autor der Headline-Klasse unter Umständen nicht mehr möglich, diese Art der Implementierung zu ändern, denn sie wird nun schon konkret von anderen Entwicklern verwendet. Und zu allen Entwicklern in einem Projektteam sagen zu müssen, dass sie ihren Code noch mal ändern müssen, weil man selbst etwas ändern möchte, kann durchaus unangenehm werden.

Schauen wir uns also an, wie man es besser machen kann. Listing 3.2 zeigt die überarbeitete Klasse.

```
package {

   import flash.geom.Point;

   public class Headline extends Sprite {

      private var xOffset:Number = 0;
      private var yOffset:Number = 0;

      public static const H1:int = 1;
      public static const H2:int = 2;
      public static const H3:int = 3;

      private static const H1_OFFSET:Point = new Point( -4, -3);
      private static const H2_OFFSET:Point = new Point( -2, -2);
      private static const H3_OFFSET:Point = new Point( 0, -1);
```

```
        function Headline() {

            // Hier nur Initialisierungen ...
        }

        public function setSize(size:int):void {

            switch(size) {
                case H1: setOffset(H1_OFFSET); break;
                case H2: setOffset(H2_OFFSET); break;
                case H3: setOffset(H3_OFFSET); break;
                default: throw new Error("Ungültige Headlinegröße");
            }
            refresh();
        }

        private function setOffset(i_Offset:Point):void {

            xOffset = i_Offset.x;
            yOffset = i_Offset.y;
        }

        private function refresh():void {

            // Aktualisiert das Layout der Headline ...
        }
    }
}
```

Listing 3.2: Beispiel für eine bessere Kapselung

In dieser Variante haben wir nun die Klasse so verändert, dass nach außen nur noch zwei Dinge sichtbar werden, nämlich zum einen die möglichen Headlinegrößen, die nun aber als Konstanten mit nichtfunktionalen Werten versehen sind (soll heißen, statt der Zahlen 1 bis 3 hätten wir auch andere Zahlen oder irgendwelche Zeichenfolgen nehmen können, es würde keinen Unterschied machen). Das ist wichtig, denn wir benutzen hier die Konstanten eigentlich als Enumeratoren, die es ja leider in ActionScript nicht gibt. Damit wir sie aber zumindest korrekt simulieren können, dürfen die in den Konstanten abgelegten Werte keinerlei funktionale Bedeutung haben. Zum anderen haben wir eine Funktion `setSize()` eingeführt, die uns erlaubt, die Größe der Headline unter Verwendung einer der drei Konstanten zu setzen. Wie wir an der Implementierung erkennen können, kann hier auch nicht geschummelt werden, denn die tatsächlichen Offsets sind private.

Außerdem haben wir die implizite Funktionalität aus dem Konstruktor entfernt. Ein Konstruktor konstruiert, das ist seine Aufgabe. Ihm weitere Aufgaben zu geben, geht gegen die Kohärenz, die wir noch in einem der nächsten Abschnitte besprechen werden.

Unsere Headline-Komponente hat nun eine klare Schnittstelle, die nur die Funktionalität nach außen verfügbar macht, die wir auch angedacht haben. Dadurch vermeiden wir Fehler durch falsche Benutzung, und wir machen unseren Entwicklerkollegen das Leben ein wenig einfacher, indem wir die Komplexität dieser Klasse nach außen verringern.

Nun stellt sich abschließend natürlich noch die Frage: »Was, wenn ein anderer Entwickler nun aber eben eine andere Headlinegröße benötigt?« In einem Projektteam, in dem die Verantwortlichkeiten auf die Teammitglieder verteilt sind, obliegt es dem Autor der Headline-Klasse zu entscheiden, wie er mit so einer Anfrage umgeht. Er kann entweder eine weitere Größe einführen oder die Schnittstelle um andere Zugriffsmethoden erweitern. Entscheidend aber ist, dass der Autor die Möglichkeit behält, Änderungen vorzunehmen, ohne dass andere Entwickler, die die Headline-Klasse schon benutzen, auch an ihren Code ran müssen. Deswegen ist das richtige Vorgehen, dass der Autor mit Bedacht seine Klasse ändert und nicht jeder andere Entwickler die Klasse auf gut Glück benutzt, wie es bei einer schlecht gekapselten Klasse möglich wäre.

Es lässt sich also sagen: Die stärkste Motivation zur Kapselung ist das Verstecken von Implementierungsdetails. Dadurch, dass wir nach außen nur preisgeben, was wir können, aber nicht, wie wir es anstellen, haben wir zum einen die Möglichkeit, die Art der Implementierung falls erforderlich zu ändern, und wir ersparen den Entwicklern, die unsere Klassen benutzen, sich mit zu vielen Informationen herumschlagen zu müssen. Dadurch erhöhen wir die Wartbarkeit und Erweiterbarkeit unseres Codes. Wichtig in dem Zusammenhang ist auch zu verstehen, dass Kapselung nicht allein über die Steuerung von Sichtbarkeit realisiert wird, sondern eben auch dadurch, dass wir statt konkreten Funktionsweisen und Datentypen abstrakte Stellvertreter einsetzen. Statt also die konkreten Offsets als öffentliche Konstanten anzugeben, haben wir abstrakte Enumeratoren eingesetzt, die keinen Rückschluss mehr darauf erlauben, dass überhaupt Offsets verwendet werden. Kapselung verbirgt seine Implementierungsdetails also nicht nur über das Verstecken, sondern auch durch das Abstrahieren. Deswegen muss die öffentlich zugängliche Methode unserer Headline auch `setSize()` und nicht `setOffset()` heißen, denn die abstrakte Idee sagt, dass wir die Größe der Headline setzen können, und die Implementierung sagt, dass wir das unter anderem durch Setzen von Offsets realisieren.

Kapselung findet natürlich nicht nur bei konkreten Klassen statt, sondern auch in größerem Maßstab. Auch ein Modul oder eine ganze Bibliothek sollte eine gewisse Kapselung besitzen, damit den Entwicklern, die dieses Modul benutzen wollen, die interne Komplexität erspart bleibt. Ein Modul sollte deswegen nur bestimmte seiner Klassen zugänglich machen, mit denen ein Entwickler auch konkret arbeiten können soll. In der Welt der Entwurfsmuster ist die Fassade ein gutes Beispiel für die Kapselung auf Modulebene. In diesem Entwurfsmuster verhindert man nämlich ganz konkret, dass externe Klassen wahllos auf die internen Klassen eines Moduls zugreifen können. Stattdessen definiert man eine klare öffentliche Schnittstelle, die Fassade, die den Zugriff auf das Modul steuert. Alle anderen modulinternen Klassen sind nun hinter dieser Schnittstelle gekapselt. Das macht das Benutzen des Moduls einfacher, denn nun muss sich ein Entwickler nur noch mit der Fassade beschäftigen und nicht mehr mit all den internen Klassen des Moduls.